Developing User Interfaces with XML: Advances on User Interface Description Languages

Satellite Workshop of Advanced Visual Interfaces 2004

Gallipoli, Italy, 25 May 2004

Editors: Kris Luyten
Marc Abrams
Jean Vanderdonckt
Quentin Limbourg

Developing User Interfaces with XML: Advances on User Interface Description Languages

Satellite Workshop of Advanced Visual Interfaces 2004

Gallipoli, Italy, 25 May 2004

Editors: Kris Luyten
Marc Abrams
Jean Vanderdonckt
Quentin Limbourg

Table of contents

Retrospective on UI Description Languages, Based on 7 years Experience with the User Interface Markup Language (UIML)....1
Marc Abrams, Jim Helms

Describing Appliance User Interfaces Abstractly with XML....9
Jeffrey Nichols, Brad A. Myers, Kevin Litwack, Michael Higgins, Joseph Hughes, Thomas K. Hariss

Practical experiences with device independent authoring concepts....17
Oskari Koskimies, Michael Wamund, Peter Wolkerstorfer, Thomas Ziegert

Dynamically generated multi-modal application interfaces – position paper....25
Stefan Kost

Extensibility and Reusability of Web User Interface Components using XICL....31
Jair C. Leite, Lirismei Gomes de Sousa

Abstract User Interface Markup Language....39
Roland A. Merrick, Brian Wood, William Krebs

Into the mangle: Software engineers run creases through a user interface metaphor....47
Simon Crowle

USIXML: A User Interface Description Language for Context-Sensitive User Interface....55
Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, Murielle Florins,Daniela Trevisan

Incorporating UIDLs into Model-Driven Development....63
Xiaoping Jia, Adam Steele

The AMF Architecture in a Multiple User Interface Generation Process....71
Kinan Samaan, Franck Tarpin-Bernard

Supporting Workflow in User Interface Description Languages....79
Nicole Stavness, Kevin Schneider

Evaluation of High-Level User Interface Description Languages for Use on Mobile and Embedded Devices....87
Jan Van den Bergh, Kris Luyten, Karin Coninx

useML: A Human-Machine Interface Description Language.....95
Detlef Zuehlke, Kizito Mukasa, Alexander Boedcher, Achim Reuther

The TERESA XML Language for the Description of Interactive Systems at MultipleAbstraction Levels.....103
Silvia Berti, Francesco Correanim, Fabio Paternò, Carmen Santoro

VRIXML: A User Interface Description Language for Virtual Environments.....111
Erwin Cuppens, Chris Raymaekers, Karin Coninx

Multimodel Dialog Description for Mobile Devices.....119
Steffen Bleul, Wolfgang Mueller, Robbie Schaefer

Extending an XML environment definition language for spoken dialogue and web-based interfaces.....127
Pablo A. Haya, Germán Montoro, Xavier Alamán, Rubén Cabello, Javier Martínez

IM^{2}L: A User Interface Description Language Supporting Electronic Annotation.....135
Daniela Fogli, Giuseppe Fresta, Andrea Marcante, Piero Mussio

Extending XML UIDLs for Multi-Device Scenarios.....143
Elmar Braun, Max Mühlhäuser

Adaptation for Device Independent Authoring.....151
Guido Menkhaus, Sebastian Fischmeister

Best of both Worlds – linking of XUL to X3USGP.....159
Andreas Müller, Peter Forbrig

Retrospective on UI Description Languages, Based on 7 Years Experience with the User Interface Markup Language (UIML)

Marc Abrams
Harmonia, Inc.
PO Box 11282
Blacksburg, VA 24062-0106, USA
+1 540 951 5901
mabrams@harmonia.com

Jim Helms
Harmonia, Inc.
PO Box 11282
Blacksburg, VA 24062-0106, USA
+1 540 951 5900
jhelms@harmonia.com

ABSTRACT
In this position paper, we reflect on the experiences of the project to develop and evolve the User Interface Markup Language (UIML), starting from its origin in 1997. We have two objectives in our discussion. First, we suggest one possible vision of what the community of researchers and practitioners working on User Interface Description Language (UIDL) could achieve. Second, we relate lessons that we learned from UIML in the hope that other UIDL researchers may avoid reinvention of the wheel.

Keywords
User Interface Description Language, UIML, XML

A Vision for the UIDL Community

Our vision for the UIDL community is to create open standard XML user interface implementation languages through the use of the best Human/Computer Interaction (HCI) research results, and to advance interoperability by using these languages in HCI tools, including web application development tools and Integrated Development Environments for programming languages. Often, good user interface development methods are defined in HCI research, but a wide gap separates the conceptualization of those methods from what HCI practitioners have available in the tools they use everyday. Practitioners need tools and technologies that span the gap between concept and realization, enabling practitioners to immediately put best practices to work.

In this paper we specifically focus on the need for an open standard XML language to express implementations of user interface (UI) designs that are multi-device, multi-lingual, multi-modal, independent of UI metaphor, and designed to interoperate with concepts and languages from OASIS, W3C, and other standards groups. Beside a XML UI implementation language, other languages are needed, for example, to represent models (e.g., ConcurTaskTree Notation [12]) and to express mappings and transformation rules between XML languages. There should be a healthy debate in our community as to what language types are needed to support the HCI software development life cycle, how the languages represented at this workshop should fit together, and how to cull the best ideas from UIDL designs.

HCI best practices can be adopted more easily by viewing HCI design and implementation as a process of transforming an initial design model to the open standard XML UI implementation language, then transforming the open standard XML UI implementation language to target languages for deployment. This belief has been borne out by our implementation experience with the User Interface Markup Language (UIML) [16]. UIML itself uses transforms (or mappings), and our UIML implementations make liberal use of transforms. Note that the target languages can be XML languages (e.g., XHTML, SVG) or programming languages (e.g., C# and Java).

Achieving our vision of language standard plus tool adoption will take years, but will unleash practitioners to focus on UI usability and quality rather than the mechanics of building UIs. Achieving the vision will also provide an easy way for HCI researchers to integrate their prototypes into development tools. For example, an HCI research project on a new model could create a prototype that exports an open standard XML modeling language, which is then imported by some tool to style UIs. This can reduce the gap between research and practical tools. Table 1 lists potential benefits from the vision.

If we continue to work in isolation, there is a danger that we will generate many good languages that are used by separate, limited user communities. If this happens, we will have created only a patchwork of interoperable solutions that fail to achieve the largest impact possible on the many people who create UIs.

Creating UIDLs that Scale to Handle All Possible UIs

There are many XML-based UIDLs. In addition to languages from W3C (HTML, XHTML, XForms, and SVG), early efforts include UIML, the XML User Interface Language (XUL) [9], the Alternate Abstract Interface Markup Language [21], and the eXtensible Interface

Markup Language (XIML) [15]. The OASIS Technical Committee investigating UIML has produced a document comparing UIML to many of these standards and describing a number of related ongoing efforts [17].

One challenge in creating UIDLs is handling the entire range of user interfaces that people build, from simple personal applications to complex enterprise applications (e.g., a ship with tens of thousands of UIs); from throw-away software with a single version (e.g., a class project) to software that is used for decades in many different versions (e.g., a banking or hospital system); from UIs deployed to one device to UIs deployed to thousands of vastly different devices (e.g., 2D or 3D, virtual or augmented reality, voice or haptic), for any UI metaphor.

On the complex end, we suggest two examples that can serve as use cases for the generality of UIDLs.

The first example is from the automotive industry. Automakers are envisioning automobiles that have XML-driven display and voice interfaces that are connected by a wireless link to a Vehicle Service Provider (VSP) [6]. General Motor's OnStar™ and Mercedes' Tele-Aid™ services are voice-only interfaces that provide a glimpse of what VSPs could offer in the future. For example, a VSP might provide voice or visual UIs to automobile occupants to reserve a hotel room, find a restaurant, order and view videos-on-demand, or provide navigation information integrated with traffic from an off-board server. A VSP must deliver UIs to a wide range of hardware and software in many vehicle makes, models, and model years. This could be done by distributing a UIDL from the VSP to all automobiles, then rendering the UIDL in the vehicles on the fly. Or, the UIDL could be rendered at the VSP into other formats (e.g., Java applets, XHTML, VoiceXML) for transmission to automobiles. Doing so requires a UIDL powerful enough to express any UI for any type of device, and designed for efficient implementation either on a high-throughput server or a low-end computer in an automobile.

The second example of complex UI environment is the maritime environment. Shipboard computers run many UIs to control everything from navigation to communications. Ships are built over many years, and even ships in the same class use different hardware and software. These are "refreshed" periodically during the ship's lifetime, resulting in a large installed base of computer systems to support. As with the automotive case, UIs for applications that run on ships of various ages could be simplified by starting with an HCI model, and then mapping the model to a family of UIs in a UIDL that can be mapped to the particular hardware and display characteristics in each ship.

Lessons for UIDL Community from UIML Experience

The experience gained since 1997 in developing UIML offers lessons for achieving the authors' vision. UIML design started in 1997, and produced three successive language versions (UIML1, 2, and 3), during which UIML evolved from an XML-compliant language to a canonical meta-language for representing any UI (these terms are discussed later). The language design goals are to subsume the expressive power of any other language used to represent UI implementations, and to permit efficient implementations that are competitive to traditional, non-XML languages used for UI implementation (e.g., C, C++).

The language was influenced by Nicholas Wirth's minimalist philosophy for language design; UIML 3.1 has just 36 tags. The "working set" of tags a UI developer uses is smaller than 36; many tags are only used to create mappings to a target widget set or the API of the business logic to which the UI is connected. Implementations, or partial implementations, of UIML have been done to platforms including Java, C, C++, C#, HTML3.2 and 4.0, WML, VoiceXML, JavaScript, and with the APIs from Swing, AWT, Qt, GTK+, PocketPC, and Palm.

The primary goal of UIML's designers has been to create an open standard for UI definition. The Organization for the Advancement of Structured Information Standards (OASIS) established a UIML Technical Committee (TC) that has examined UIML and is in the process of evaluating language issues. This TC is available to help the community design and standardize an open UIDL. The TC is open to all to shape the goals of the new language, and could even restart on a new path, if needed. While the TC is currently focused on UIML, contributors who have worked with other UIDLs are welcome.

Key UIDL Elements: Findings from UIML Experience

Through our experiences working with UIML, we have identified key elements that must be present in a UIDL.

The UIDL must be a meta-language.

Consider XML. After years of battles to standardize new versions of HTML, people quickly recognized the need for a meta-language, in which the tags are defined outside the specification. XML cannot be used by itself, but rather must be used in conjunction with a vocabulary of element and attribute names (from a DTD or XML schema). In this way XML could be standardized once and then "extended" an infinite number of times through new vocabularies. Under this situation, tools could be built once and used for many different XML-based languages.

The XML lesson is applied in UIML. Rather than defining tags that are specific to a particular metaphor (e.g, <window>), UIML uses about two dozen generic tags. The most important are `<part>` and `<property>`. A UI comprises a set of parts, hence the `<part>` tag. Every part in a UI has presentation properties, hence the `<property>` tag. As with XML, UIML cannot be used by itself; one must define a

vocabulary of part classes and property names. Vocabularies have been developed for AWT, Swing, WML, HTML, PalmOS, VoiceXML, and QT. In addition, work is under way to define a generic vocabulary and rule-based approach to transformation to map the generic vocabulary to platform-specific vocabularies [3].

Through definition of new vocabularies, UIML is infinitely extendable to new devices, UI languages, and UI metaphors that develop over time.

Another benefit of using vocabularies is to support multiple abstraction levels with a single UIDL. These abstractions can range from capturing author-intent and early design model representations to widget set specific abstractions. Each set of abstractions would reside in their own vocabulary. Transforms and mappings can then be established between the various vocabularies to automate the process of carrying a UI design from a high-level abstraction down to a target language representation.

The UIDL must support model-based UI development.

The automotive and maritime UI examples sited earlier represent complex UIs that are modified over a long life time and deployed to many types of devices. Model-based UI development is almost essential to cope with the complexity. For example, the operational role of a driver in a car or a sailor on a ship can be described in task models or activity diagrams, and these are refined through a process into the physical UIs that people use.

There are two approaches to creating UIDLs. The first is to create a single language that can represent both model and interface design, and can be mapped to any platform or device from a single interface description. The second is to use multiple languages, such as a task model language, a user model language, and a UIDL to specify interface parts and presentation properties that are derived from the task and user models. Obviously the languages must fit together. The multiple-language approach is used in TERESA [13], and is the approach we use with UIML.

At first glance it may not be clear how UIML would fit other languages. The key is to use UIML vocabularies to represent different levels of abstraction. Two projects, one at Virginia Tech [5] and the second at Harmonia, have explored mapping task models (in ConcurTaskTree notation) to a *logical* UI description expressed in UIML using a vocabulary consisting of domain-specific *abstract* parts. The logical UI is then mapped to a *physical* UI by mapping domain-specific logical parts to physical parts (e.g., Java Swing classes or HTML tags) and adding presentation style. The physical UI definition is also expressed in UIML, but with a completely different, concrete vocabulary. Figure 1 summarizes the process.

The UIDL must be a canonical language.

UIML seeks to create a canonical syntax that can represent the superset of all concepts represented by other languages for UI implementation. Doing so would simplify many things. Individuals writing UI code would just learn one syntax, UIML, rather than mastering the idiosyncrasies of different target UI languages. UI implementers would be insulated from the frequent updates to target languages. Instead, the tools to map UIML to target languages would be implement language updates, and only tool implementers would need to be concerned with details of how to use the new features programmatically.

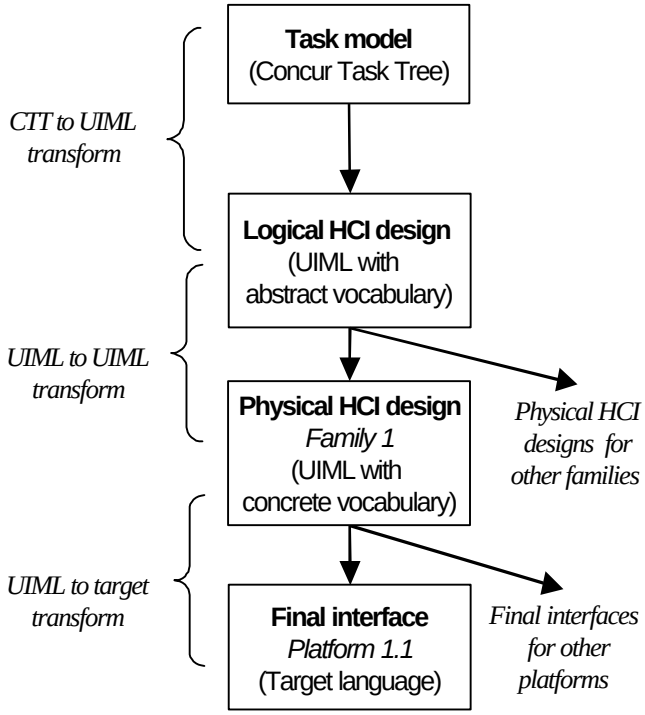


Figure 1: Transforming task models to UIML interface descriptions.

Also, multi-platform authoring tools would only need to write one syntax (UIML), which in a separate step could be mapped to any target UI language. Moreover, if all UI development tools used this syntax as an intermediate UI representation, then a design created with one tool could be imported into another tool. With converters written to go from any UI language into UIML, and from UIML to any UI language, one could create a UI with a Web page design tool, save it to UIML, then import the UIML into a Java interface builder. In this environment, the problem of interoperability is reduced to merely resolving differences in widget sets, which is a much smaller task than having to rewrite the UI in a different language.

A UIDL must separate concerns.

In order for a UIDL to effectively define interfaces for any platform or device it must overcome the MxN problem

described in [8]. Overcoming the MxN problem requires separating out much of the UI definition into re-usable chunks. These chunks follow a pattern that looks much like the Model View Controller (MVC) design pattern.

Defining a UI in UIML answers the following six questions.

- What parts comprise the UI? (the <structure>)
- What presentation style for each part? (the <style>)
- What content for each part? (the <content>)
- What behavior do parts have? (the <behavior>)
- How to connect to outside world? (the <logic>)
- How to map to target UI toolkit? (the <presentation>)

UIML follows a variant of the MVC design pattern, the Meta-Interface Model [13], shown in Figure 2.

The UIDL must identify "presentation logic" as first class entity, and represent it in a portable, device-independent manner to allow compilation to target language.

How should user interaction with the UI be represented? How should the interconnection of the UI to external entities (e.g., business logic, databases, objects, remote procedures) be represented? The landscape of UIDLs contains remarkably varied answer to these questions. Added to this is the variety of ways imperative languages answer these questions. For example, the behavior that results from a button click can be represented in these ways: in HTML forms, either an attribute (action) in a tag (form) or via scripting; in Java JFC, a method call (actionEvent) in an object implementing an interface (ActionListener); and in WML, nested tags (<go> nested in <do>).

In UIML, the <behavior> element provides a single syntax to represent events for any target UI language. Drawing from rule-based systems, the <behavior> element contains a set of rules, each with a condition and an action. When the condition holds, the action is performed. Actions can set properties or call external methods. Conditions are based on events; Green showed that the descriptive power of events is greater than other methods (state transition networks and context-free grammars) [11]. For example, a cut-and-paste operation can be described with events but not the other methods.

As was the case with UI `<part>`s, events also are treated as class names, and thus their vocabulary is defined outside UIML. The `<d-class>` element specifies the mapping of event classes to a class or tags in the target UI language. For example, a "selection" event class might be defined, mapping to a java.awt.event.ActionEvent in Java, but to OnClick in HTML. This allows UIML to be used with various types of UI metaphors and devices.

Suppose one wants to create a UI that calls CORBA objects and queries LDAP servers to provide the UI content. Suppose further that the UI will be deployed to Java, HTML, and WML. Then the description of how to connect the UI to CORBA and LDAP would be described once, and the rendering engines for Java, HTML, and WML would figure out how to achieve this (e.g., by using HTTP POST along with a proxy to allow HTML to call CORBA or query LDAP). UIML provides a `<call>` element to describe the actual connection to the entity outside the UI, and another `<d-class>` element defines how to resolve the `<call>` by giving the declaration of an external object or service interface.

A UIDL is Not a Silver Bullet: It Must Be Augmented by Architecture, Methodology, and Tools

People building multi-device UIs who contact us with questions about UIML are often looking for a "silver bullet": a way to describe a UI once, then magically transformed the description to a highly usable experience on any target device, from desktop PC to small mobile phone. We call this the "silver bullet fallacy": the organization of UIs varies radically from device to device, based on decisions ultimately made by human designers, and those steps that can be automated (e.g., reformatting based on small variations in screen size) are done by software transformations that are part of the architecture used to implement UIML [4].

A UIDL *can* be sufficiently powerful to *express* the collection of implementations needed on a wide range of devices in a concise fashion, but a methodology is needed to *derive* those implementations. For example, UIML represents a family of UIs in a tree structure. Items close to the root of the tree are shared by all devices (e.g., the need to a UI control to perform a search operation). Items close to the leaves of the tree represent differences among devices (e.g., use of a specific HTML tag on a phone display, versus use of a specific C++ widget on a desktop computer). However, generation of the tree is a job for a tool that is based on an HCI development methodology.

What is Hard to Get "Right" in a UIDL

In the creation and implementation of the UIML specification, we discovered several issues that are challenging to address in a UIDL. These issues represent areas of UI design that are necessary to represent in a UIDL so that the UIDL provides a complete and useful description of the UI. In this section we discuss each issue and, where available, provide the solution we designed for UIML. Each of the solutions presented represent one method of overcoming the challenges provided by these issues, but do not necessarily represent the only or best way of accomplishing the goals of the language.

Issue 1: Dynamic UIs

In order for a UIDL to fully describe a UI and its behavior, the UIDL must be able to represent the dynamic aspect of the user interaction. This dynamism comes in two forms: properties of the UI must be able to be dynamically set from the application's presentation logic; and constituent parts of the UI must themselves be interchangeable so that the interface can change through out the user's interaction.

UIML addresses the first of these two by specifying a `<call>` element. `<call>` abstracts any invocation of code (that uses a language other than UIML). This tag is used to access presentation logic in the application and return the value generated by the backend code. Values returned from the presentation logic can then be used to set property values on the UI. The example below shows how a `<call>` can be used to set the value of a `<property>`.

```
<property part-name="text" part-
name="label">
  <call name="back1.getText"/>
</property>
```

The `<d-component>` element defining back1 is this:

```
<logic>
  <d-component id="back1"
        maps-to="org.uiml.example.myClass">
    <d-method id="getText"
        maps-to="myFunction"
        returns-type="String"/>
  </d-component>
</logic>
```

Rearranging the structure of the UI is more challenging than simply calling a segment of code in the presentation logic. Traditional UIDLs excel at defining the static, initial structure of the UI. For a UIDL to completely describe the user's interaction with the interface, it must be able to describe changes in the physical structure of the UI.

In UIML, there is the notion of a UI tree that is represented in the `<structure>` element. This tree comprises `<part>` elements and their containership relations. UIML specifies a `<restructure>` tag that modifies the UI tree, replacing or adding branches. `<template>` elements are used to describe the new branches, and the `<restructure>` tag indicates where and how the branch should be added to the tree. Consider the UI described in the `<structure>` below.

```
<structure>
  <part class="JFrame"         id="F">
    <part class="JButton"      id="B"/>
    <part class="JPanel"       id="A">
      <part class="JLabel"     id="L1"/>
      <part class="JTextField" id="TF"/>
      <part class="JCheckbox"  id="C"/>
    </part>
  </part>
</structure>
```

To replace the contents of the panel with just a label and a text field, one would do this:

```
<restructure at-part="A" how="replace">
  <template id="T5>
    <part>
      <part class="JLabel"     id="L2"/>
      <part class="JTextField" id="TF2"/>
    </part>
  </template>
</restructure>
```

The set of children of A in the virtual UI tree is replaced with the set of children listed in the restructure tag. So now the panel has only L2 and TF2 from the template as children. Figure 3 shows the state of the UI tree during the processing of a `<restructure>`.

Issue 2: Reusable UIs

One of the goals of the UIDL community is to improve the reusability of UI designs. However, facilitating reusability as a first class notion in a UIDL requires careful consideration. Should the language have objects? If so, does it need to be completely object-oriented or object-based? Do other paradigms provide equivalent re-use?

UIML was designed with re-use in mind. `<template>` elements contain chunks of UIML description that can be sourced by other UIML documents. The `<template>` element permits shortcuts when writing UIML. It allows

- one fragment of UIML to be inserted in multiple places in a UIML document,
- one UIML document to include a UIML fragment from another document, and
- cascading style and other elements, in a manner analogous to the CSS specification [10].

`<template>` elements work as follows. Most UIML elements can contain the `source` attribute; let such an element be *E*. The *source* attribute names a `<template>` element (either within the same document or in another document). The `<template>` element named must be an element of the same type as the element *E* (i.e., have the same tag name). The `source` attribute causes the body of the element inside the `<template>` to be combined with the body of *E*.

For example, vocabularies are often written as a `<template>` so that they may be re-used between UIML files. The example below shows how this works.

```
<peers>
  <presentation
    base="Java_1.3_Harmonia_1.0"
    source="http://uiml.org/toolkits/
    Java_1.3_Harmonia_1.0.uiml#vocab"/>
</peers>
```

In relation to this, some UIML implementers have proposed a mechanism to control which elements in a `<template>` can be overridden using the `how="cascade"` attribute. This would define certain

<template> elements as immutable so that safety- and mission-critical UI elements are always present.

A limited form of this functionality is already described in the UIML 3.x specification. The export attribute can be set on <part> and <property> elements within a template to prevent UIML authors from changing properties on a given <part>. No current mechanism is in place that would prevent a UIML author from replacing a <part> within the <template> with one of his or her own.

Issue 3: Efficiency

UIDLs by their nature add a step to the process of displaying a UI. The UIDL must be transformed into some appropriate form for display on a particular device. Typically this involves converting XML to a high-level programming language (e.g. Java or C++) or a mark-up language (e.g HTML, VoiceXML). Such conversions can be involved and computationally expensive. It is better to avoid this step when the UI is presented.

Harmonia's UIML to HTML renderer implements a facility to avoid the conversion step. Like Java Server Page (JSP) servers, Harmonia's renderer compiles the UIML into a Java executable file. The executable can be cached, and whenever the UI is needed it can be loaded from the executable file, avoiding the lengthy parsing process. The UIML definition then becomes the master file that describes the interface; it is re-parsed and rendered only when the definition changes.

Issue 4: Enforcing Separation of the Presentation Layer from the Application Logic

UIDLs describe the presentation layer of a system. Typically this includes some combination of UI description and presentation logic that drives the UI. The challenge that UIDL designers face is how to separate these two aspects of the presentation layer description. UIML does this by separating the mappings to the presentation logic into the <logic> element, and then referring to them indirectly from the interface description through <call> elements. This separation ensures that the presentation logic is removed from the UI description, allowing system architects to easily replace or update either the UI or the presentation logic in isolation.

Issue 5: Representing Different Layers of Abstraction

In order to describe portable UIs that can be deployed to multiple platforms or devices, a UIDL cannot rely on using constructs that adhere to a single UI metaphor or toolkit. For example, it would be difficult to take a UI that is described in a Java and deploy it to a voice only device. Instead a higher-level description of the UI must be available that can be mapped to a more specific widget set. Thus a UIDL must be able to represent various layers of abstraction and avoid adhering to a specific UI metaphor.

UIML accomplishes this through the use of vocabularies. UIML is a meta-language, meaning that UIML by itself cannot completely describe a UI. Instead, UIML is combined with a vocabulary of abstractions that provide class types and properties that can be applied to the parts in an interface. Vocabularies can be written to any level of abstraction for any problem domain, allowing designers to choose an appropriate set of UI abstractions to describe their current interface. Tools that use UIML with a particular vocabulary only have to be able to map the vocabulary abstractions to widgets on a given deployment platform. Thus two tools could take the same UI description and render it to different platforms via a different set of rules and transforms.

Vocabularies have the added benefit of making UIML extensible for future UI technologies.

Issue 6: Description of Iterative Entities

One difficulty of describing a UI is that the number of elements needed may not be known at design time. For example, the number of elements needed in a table that displays database query results depends on the size of the result set. A UIDL needs a mechanism for describing repetitive structures once and letting the rendering software handle the repetition.

UIML uses a <repeat> element for this. A <repeat> element encloses one <iterator> element and a set of one or more <part> elements. The <part> elements denoted as children of the <repeat> element are repeated with their children a number of times designated by the <iterator> element. The <repeat> element's parent <part> element is not repeated. The <iterator> can be used in <property> tags to set the <property> equal to the current iteration of the loop.

```
<uiml>
  ...
  <part class=”JDialog”>
    <repeat>
      <iterator id=”i”>10</iterator>
      <part class = “JCheckBox”>
        <style>
          <propertymname=”text”>
            <iterator id=”i”/>
          </property>
        </style>
      </part>
    </repeat>
  </part>
  ...
</uiml>
```

The example above shows the uses of the <iterator> element to create a JDialog containing ten JCheckBoxes, numbered 1 to 10.

Concluding Remarks

The User Interface Markup Language (UIML) is based on the concept of using transforms and mappings to extend its utility to any UI technology or toolkit. The goal of UIML is to remove the complexity of generating the UI description and focus on defining the mappings and transforms that enable UIML to be converted into the appropriate deployment language. We believe that the lessons learned in designing and implementing UIML can extend to any UIDL, and should be considered as the community seeks to create a shared UIDL.

Feedback from customers indicates that adding layers of abstraction can be more trouble than it is worth for simple UIs, but when the systems and interfaces are complex, the ability to quickly and easily modify and re-generate code becomes very important. Imagine two systems: one consists of a single dialog box while the other consists of hundreds of such windows. Now imagine the relative cost of modifying one property within each dialog of the systems. What becomes apparent is that while modifying one dialog at the source code level is practical, the opposite is true for trying to maintain a large system at this level. Here is where a UIDL can be very effective. It provides a way for non-programmers to take part in the maintenance of the system at a lower overall cost to the development effort. It also opens the possibility of structuring the UI description in such a way that each can use centralized stylesheets and property definitions, further reducing the time required to modify the interface.

The set of computing platforms and devices is too diverse to be covered by a single UIDL that relies on platform-specific constructs to describe all possible platforms and devices. A meta-language approach is essential to creating a viable, platform-independent UIDL.

In addition, we invite the formation of a working group by anyone interested in evolving a set of open standard XML languages for HCI.

ACKNOWLEDGMENTS

This material is based upon work supported by the Naval Surface Warfare Center, Dahlgren Division, under Contract No. N00421-04-C-0030.

REFERENCES

1. M. Abrams, C. Phanouriou, A. L. Batongbacal, S. M. Williams, J. E. Shuster, "UIML: An Appliance-independent User Interface Language," *Computer Networks 31*, 1999, 1695-1708; also appeared in *Proc. 8th International World Wide Web Conference*, Toronto, May 1999.
2. M. Abrams, C. Phanouriou, "UIML: An XML Language for Building Device-Independent User Interfaces," *Proc. XML 99*, Philadelphia," Dec. 1999.
3. M. F. Ali, M. Abrams, "Simplifying Construction of Cross-Platform User Interfaces using UIML," *Proceedings of UIML 2001*, Paris, France.
4. M. F. Ali, M, Pérez-Quiñones, M. Abrams, E. Shell, *Building Multi-Platform User Interfaces with UIML*, CADUI, Valenciennes, France, May 2002.
5. M. F. Ali, M. A. Pérez-Quiñones, and M. Abrams, "Building Multi-Platform User Interfaces with UIML", in *Multiple User Interfaces: Cross-Platform Applications and Context-Aware Interfaces.* ed. H. Javahery and A. Seffah, pp. 95-116. John Wiley & Sons, Ltd. 2004.
6. Automotive Multimedia Interface Consortium home page, www.ami-c.org.
7. P. Azevedo, R. Merrick, and D. Roberts."OVID to AUIML - User-Oriented Interface Modelling." *Paper presented at TUPIS 2000*.
8. R. B. Case, S. H. Maes and T. V. Raman, "Position paper for the W3C/WAP Workshop on the Web Device Independent Authoring", *W3C Workshop on Web Device Independent Authoring*, Bristol, England. October 2000.
9. T. Cheng. "XUL: Creating Localizable XML GUI". *In Proceedings of the Fifteenth Unicode Conference.* 1999.
10. B. Bos, H. W. Lie, C. Lilley, and I. Jacobs, *Cascading Style Sheets, level 2*, CSS2 Specification. W3C Recommendation 12-May-1998, http://www.w3.org/-TR/REC-CSS2/.
11. Green, M. A Survey of Three Dialogue Models. In *ACM Transactions on Graphics*, vol 5, no. 4, July 1986, pp. 244-275.
12. F. Paternò, C. Mancini, and S. Meniconi. "Concur-TaskTrees: A Diagrammatic Notation for Specifying Task Models" *In Proceedings Interact'97*, Chapman & Hall, July 1997. pp.362-369.
13. F. Paternò, F., C. Santoro, A Unified Method for Designing Interactive Systems Adaptable to Mobile and Stationary Platforms, *Interacting with Computers 15* (2003), Elsevier. pp. 349-366.
14. C. Phanouriou, "UIML: A Device-Independent User Interface Markup Language", Ph.D. dissertation, September 2000, Computer Science Dept., Virginia Tech. http://scholar.lib.vt.edu/theses/default.htm
15. A. Puerta, and J. Eisenstein, "XIML: A Multiple User Interface Representation Framework for Industry", in *Multiple User Interfaces: Cross-Platform Applications and Context-Aware Interfaces.* ed. H. Javahery and A. Seffah, pp. 119-148. John Wiley & Sons, Ltd., 2004.
16. *User Interface Markup Language Technical Committee home page*, Organization for the Advancement of

Structured Information Standards, www.oasis-open.org/committees/tc_home.php?wg_abbrev=uiml.

17. OASIS User Interface Technical Committee, "The Relationship of the UIML 3.0 Spec. to Other Standards/Working Groups". Available at: http://www.oasis-open.org/committees/documents.-php?wg_abbrev=uiml, September 8, 2003.
18. World Wide Web Consortium, http://www.w3c.org/
19. ximl.org, http://www.ximl.org/
20. *The XML User Interface Language (XUL) Project*, http://www.mozilla.org/projects/xul/
21. G. Zimmermann, G. Vanderheiden, and A. Gilman. "Universal Remote Console Prototyping of an Emerging XML Based Alternate User Interface Access Standard." Proc. of *11th Int. WWW Conf.*

Creating UIDLs With These Features…	*Could Produce These Benefits…*
Use one common UIDL for interoperability of tools (e.g,. Rapid prototyping/GUI design, software test, transformation)	HCI practitioners can • Save time • Choose best tool for job
UIDL-compatible tools can be plugged together with new tools	Help HCI researchers evaluate prototypes from research by plugging them into production tools and motivate people to create better HCI tools & techniques
Single UIDL can be used throughout software life cycle	Ability to reduce software life cycle cost from rapid prototype to implementation to years of maintenance
Software developers create the future "legacy" applications using UIDL	Facilitate integration and UI reengineering for legacy applications; integrate separate legacy apps with new UIs with common look/feel
Facilitates adaptation of UIs to current & future devices, even in long-life systems	Properly designed UIDL and tools allow adaptation and re-mapping to future devices 20 years from now
Facilitates UI accessibility, personalization	UI is put in highly malleable format

Table 1. Benefits of Vision for Open Standard UIDLs

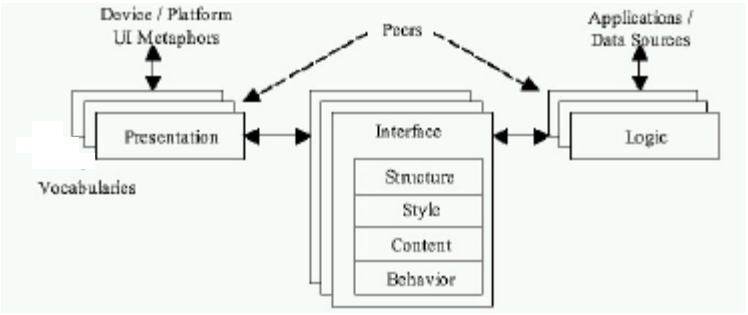


Figure 2. Meta-Interface Model, a Generalization of MVC

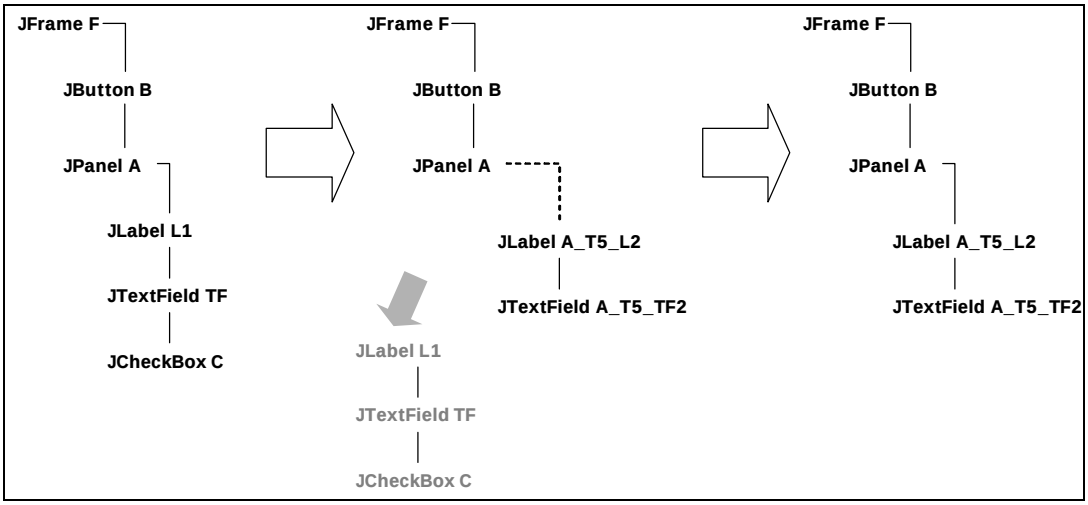


Figure 3: The state of the UI tree before, during, and after the <restructure> is processed.

Describing Appliance User Interfaces Abstractly with XML

Jeffrey Nichols[*], Brad A. Myers[*], Kevin Litwack[*], Michael Higgins[†], Joseph Hughes[†], Thomas K. Harris[*]

[*]School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
{jeffreyn, bam, klitwack, tkharris}@cs.cmu.edu
http://www.pebbles.hcii.cmu.edu/puc/

[†]MAYA Design, Inc.
Suite 702
2100 Wharton Street
Pittsburgh, PA 15203
{higgins, hughes}@maya.com

ABSTRACT

This paper describes an XML-based language for describing the functions of appliances, such as televisions, VCRs, copiers, microwave ovens, and even manufacturing equipment. Our description language is designed to be concise, easy to use, and contain no presentation information. It has been used to describe more than twenty diverse appliances. The functional descriptions written in our language are used to automatically generate remote control interfaces for appliances. We have used these descriptions to generate both graphical and speech interfaces on handheld computers, mobile phones, and desktop computers.

Keywords

User interface description language (UIDL), automatic interface generation, remote control, appliances, personal digital assistants (PDAs), handheld computers, Pebbles, personal universal controller (PUC)

INTRODUCTION

It has long been a goal of researchers to develop a user interface description language (UIDL) that can describe a user interface without resorting to low-level code. A UIDL can reduce the amount of time and effort needed to make user interfaces by providing useful abstractions and supporting automation of the design process. For example, this might allow the same interface description to be rendered on multiple platforms. In this case, a UIDL is particularly beneficial because most of the code and implementation time is spent on the user interfaces in today's desktop applications. Without a multi-platform UIDL solution, even more will be required as applications become more distributed and user interfaces to those applications are needed on multiple platforms.

We are developing a UIDL for describing appliance user interfaces as part of our work on the *personal universal controller* (PUC) project [7]. The goal of the project is to provide users with user interface devices that can remotely control all of the appliances in the users' environments. We imagine that these user interface devices would use a variety of platforms, including handheld devices with graphical interfaces and hidden PCs with speech recognition software. To remotely control an appliance, the user interface device engages in two-way communication with the appliance, first downloading a description of the appliance's functions written in our UIDL, and then automatically creating a high-quality interface. The device sends control signals to the appliance as the user interacts with the interface, and also receives feedback on the changing state of the appliance.

The UIDL that we have designed, which we often refer to as our appliance specification language or just specification language, is a very important part of the PUC system. Not only must it describe the appliance in sufficient detail for the interface generators to create a high-quality interface, but it must be concise and short enough to be efficiently transmitted across wireless networks and parsed by embedded computing devices. Our specification language must also be descriptive enough to cover the complete functionality of any appliance, so that a PUC device can generate a complete user interface. Finally, the language needs to be abstract enough so that interfaces can be generated on multiple platforms from the same appliance specification.

This paper starts by discussing related work, both to our UIDL and the PUC system as a whole. Then we elaborate on the design principles for our UIDL, followed by a description of a study we conducted to inform our design. Then the language is described in detail, followed by some brief analysis of the strengths and weaknesses we have found in the current design.[‡]

RELATED WORK

A number of research groups are working on controlling appliances from handheld devices. Hodes, *et. al.* propose a similar idea to our PUC, which they call a "universal interactor" that can adapt itself to control many devices [4]. However, their research focuses on the system and infra-

Submitted for Publication

[‡] Portions of this paper are adapted from previously published material in [7] and [9].

structure issues rather than how to create the user interfaces. An IBM project [3] describes a "Universal Information Appliance" (UIA) that might be implemented on a PDA. The UIA uses an XML-based language called MoDAL from which it creates a user interface panel for accessing information. However, the MoDAL processor apparently only handles simple layouts and its only type of input control is text strings. The Stanford ICrafter [12] is a framework for distributing appliance interfaces to many different controlling devices. While their framework supports the automatic generation of interfaces, their paper focuses on hand-generated interfaces and shows only one simple automatically generated interface. They also mention the difficulty of generating speech interfaces.

The Xweb system [10] is an infrastructure that supports automatic generation of user interfaces from abstract descriptions, and supports multiple generation platforms, including speech. The PUC extends these ideas by adding more detail in the specification language and supporting more features in the automatic interface generation process.

The INCITS/V2 [20] standardization effort is developing the Alternative Interface Access Protocol (AIAP) to help disabled people use everyday appliances with an approach similar to the PUC. AIAP contains a description language for appliances that different interface generators use to create interfaces for both common devices, like the PocketPC, and specialized devices, such as an interactive braille pad. We are collaborating with the V2 group and they have incorporated many of our language ideas into their standard.

A number of research systems have looked at automatic design of user interfaces for conventional applications. These sometimes went under the name of "model-based" techniques [16]. Here, the programmer provides a specification ("model") of the properties of the application, along with specifications of the user and the display. Of particular note are the layout algorithms in the DON [5] and TRIDENT [18] systems that achieved pleasing, compact, and logical placements of controls. We extend these results to create panels of controls on significantly different handhelds, without requiring designer intervention after the interfaces are generated.

UIML [1] is an XML language that is designed to provide a highly-device independent method for user interface design. UIML differs from the PUC in its tight coupling with the interface. Designers using UIML can define the types of components to use in an interface and the code to execute when events occur. The PUC specification language leaves these decisions up to each platform's interface generator.

Recently a new general purpose language has been introduced for storing and manipulating interaction data. The eXtensible Interface Markup Language (XIML) [13] is an XML-based language that is capable of storing most kinds of interaction data, including the types of data stored in the application, task, and presentation models of other model-based systems. XIML was developed by RedWhale Software and is being used to support that company's user interface consulting work. They have shown that the language is useful for porting applications across different platforms and storing information from all aspects of a user interface design project.

It is common to find task information included in abstract UIDLs, though the PUC language does not yet include such information. Paterno's ConcurTaskTrees [11] is one such language, which has a graphical notation and allows the specification of concurrent tasks (not possible in earlier languages). ConcurTaskTrees also allows the specification of who/what is performing the task, whether it be the user, the system, or an interaction between the two.

DESIGN PRINCIPLES

Before and during the design of the specification language, we developed a set of requirements and principles on which to base our design [8]. Some of the principles are:

Descriptive enough for any appliance, but not necessarily able to describe a full application. We found that we were able to specify the functions of an appliance without including some types of modeling information that earlier systems included, such as task models and presentation models. This is possible because appliance interfaces almost always have fewer functions than a typical application, and rarely need direct manipulation techniques in their interfaces.

Sufficient detail to generate a high-quality interface. We conducted a user study to determine how much detail would be needed in our specification language. Note that this principle is different than the first. It would have been possible to completely describe the appliance without readable labels or adequate grouping information that is needed for generating a good user interface. For example, the Universal Plug and Play (UPnP) standard [17] includes an appliance description language that does not include sufficient detail for generating a good user interface.

No specific layout information should be included in the specification language. We wanted to ensure that our language would be general enough to work for interface generators running on a wide-variety of platforms. Another solution for addressing the multi-platform problem is to include multiple concrete interface descriptions in the appliance specification (the INCITS/V2 standard [20] and ICrafter [12] support this approach). We chose not to take this approach, because it does not support future platforms that cannot be anticipated at design time. This approach also makes it difficult to support many of the expected benefits of automatically generating interfaces, such as adaptation and personalization.

Short and concise are very important principles for the design of our language. Appliance specifications must be sent over wireless networks and processed by computing devices that lack the power of today's desktop machines.

To ensure performance is adequate, the specification language must be concise. Why then choose a verbose format like XML as the basis for our language? We chose XML because it was easy to parse and there were several available parsers. XML is also a very compressible format, which can reduce the cost of sending specifications over the network, though our system does not use any compression.

Only one way to specify any feature of the appliance is allowed in our specification language. This principle makes our language easy to author and easy to process by the interface generator. It also makes it impossible for an author to influence the look and feel of user interfaces by writing their specification in a particular way. Some examples of design choices influenced by this principle are shown later.

PRELIMINARY USER STUDIES

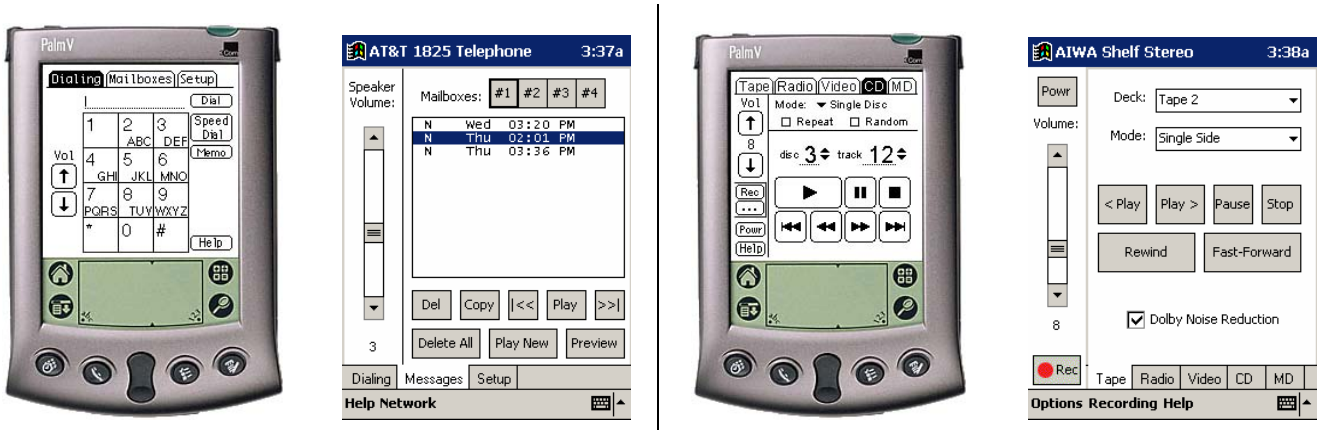
These principles guide the design our language, but do not suggest what information should be included or what level of detail is needed to automatically generate a high-quality interface. In order to determine what content should be included in a specification, we hand-designed several remote control interfaces for existing appliances. Then user studies were conducted to compare the hand-designed interfaces to the manufacturers' interfaces (described in more detail in [6]). This approach allowed us to concentrate on the functional information that should be included as content in the specification language. It also showed that a PUC device could be easier to use than interfaces on actual appliances.

We chose to focus on two common appliances for our hand-designed interfaces: the Aiwa CX-NMT70 shelf stereo with its remote control, and the AT&T 1825 telephone/digital answering machine. We chose these two appliances because both are common, readily available, and combine several functions into a single unit. The first author owns the Aiwa shelf stereo that we used, and the AT&T telephone is the standard unit installed in many offices at Carnegie Mellon. Aiwa-brand stereos seem to be particularly common (at least among our subject population) because ten of our twenty-five subjects owned Aiwa systems.

We created our hand-designed interfaces in two phases, initially on paper for the Palm platform and later as Visual Basic implementations on a Microsoft PocketPC (see Figure 1). Each interface supported the complete set of appliance functions. At each phase, we iteratively improved the interfaces with heuristic analyses and performed a user study. The user study in each phase was dual-purpose: to compare our hand-designed interfaces with the interfaces on the actual appliances and to find problems in the hand-designed interfaces.

The comparison study in both phases showed that our hand-designed interfaces were much better than the manufacturer's interfaces on the actual appliances [6]. In both studies users were asked to perform a variety of simple and complex tasks. Some simple tasks were dialing the phone or changing the volume on the stereo, whereas some complex tasks were programming a list of tracks into the stereo's CD player or copying a message between two of the four mailboxes on the telephone's built-in answering machine. We found that for both hand-designed interfaces, Palm paper prototypes and PocketPC implementations, users completed tasks in one-half the time and with one-half the errors as compared to the actual appliances [6].

The large differences in this study can be attributed to problems with the appliance interfaces. Most of the problems users had with the built-in appliance interfaces could be traced to poor button labels and inadequate interface feedback. Both appliances had buttons with two functions, one when the button was pressed and released and one when the button was pressed and held. Our subjects rarely discovered the press and hold function. The stereo also had buttons that changed function with the appliance's mode.



(a) (b) (c) (d)

Figure 1. Hand-designed interfaces for the phone (a-b) and stereo (c-d) on the Palm and PocketPC. The Palm interfaces are paper prototypes, whereas the PocketPC interfaces were actually implemented in Microsoft's embedded Visual Basic.

Interface Analysis

Once we were confident that our interfaces were usable, we analyzed them to understand what *functional* information about the appliance was needed for designing the interfaces. This included questions such as "why are these elements grouped together?" or "why are these widgets never shown at the same time?" These are questions that might suggest what information should be contained in the specification language.

As we intuitively expected, grouping information was very important for our hand-designed interfaces. We noted that grouping information could generally be specified as a tree, and that the same tree could be used for interfaces of many different physical sizes. User interfaces designed for small screens would need every branch in the tree, whereas large screen interfaces might ignore some deeper branches.

We also found that grouping is influenced by modes. For example, the Aiwa shelf stereo has a mode that determines which of its components is playing audio. Only one component can play at a time. In the stereo interfaces shown in Figure 1c-d you will note that a tabbed interface is used to overlap the controls for the CD player, tape player, etc. Other controls that are independent of mode, such as volume, are available in the sidebar. Unlike regular grouping information, information about modes gives explicit ideas about how the user interface should be structured. If two sets of controls cannot be available at the same time because of a mode, they should probably be placed on overlapping panels. We designed dependency equations to describe appliance mode information in our language.

We also noticed that most of the functions of an appliance were manipulating some data in a definable way, but some were not. For example, the tuning function of a radio is manipulating the current value of the radio station by a predefined increment. The seek function also manipulates the radio station value, by changing it to the value of the next radio station with clear reception. This manipulation is not something that can be defined based on the value of a variable, and thus it would need to be represented differently in our language.

Each of our interfaces used different labels for some functions. For example, the Palm stereo interface (see Figure 1c) used the label "Vol" to refer to volume, whereas the PocketPC stereo interface (see Figure 1d) used "Volume." We expected that this problem would be even worse for much smaller devices, such as mobile phones or wristwatches. Thus we felt it would be important for our specification language to include multiple labels that an interface generator could choose between when designing its layouts.

Finally, we found that all of our interfaces used some "conventional" designs that would be difficult to specify in any language. At least one example of a conventional design can be found in each of the panes in Figure 1: (a) shows a telephone keypad layout, (b) uses standard icons for previous track and next track, (c) shows the standard layouts and icons for play buttons on a CD player, and (d) uses the standard red circle icon for record. We recently developed a solution for addressing this problem called Smart Templates [9], which will be discussed in the next section.

SPECIFICATION LANGUAGE

The PUC specification language is XML-based and includes all of the information that we found in our analysis of the hand-designed interfaces. This section describes the features of our language and shows examples of how each feature is used. A language reference can be downloaded from our project web site:
http://www.cs.cmu.edu/~pebbles/puc/specification.html

State Variables, Commands, and Explanations

Our specification language supports three primitive elements for describing the functions of an appliance. We discovered from our PocketPC implementations that most appliance functions could be represented as state variables. Each state variable has a given type that tells the interface generator how it can be manipulated. For example, the radio station state variable has a numeric type, and the interface generator can infer the tuning function because it knows how to manipulate a numeric type. Other state variables include the current track of the CD player and the status of the tape player (stop, play, fast-fwd, etc.).

As mentioned above, we discovered that state variables are not sufficient for describing all of the functions of an appliance, such as the seek button on a radio. The seek function must be represented as a *command*, a function whose result cannot be described easily in the specification. Figure 2 shows examples of both state variables and commands.

Commands are also useful when an appliance is unable to provide state information to the controller, either by manufacturer choice or a hardware limitation of the appliance. For example, up and down commands might be used for volume if the appliance cannot support an integer-typed state variable. In fact, the remote control technology of today can be simulated on the PUC by writing a specification that includes only commands. Any state information must then be shown on the appliance's front panel.

Our specification language also has a feature called an explanation. Explanations are static labels that are important enough to be explicitly mentioned in the user interface, but are not related to any existing state variable or command. For example, an explanation is used in one specification of a shelf stereo to explain the Auxiliary audio mode to the user. The mode has no user controls, and the explanation is used to explain this. Explanations are used very rarely in the specifications that we have written.

Type Information

Each state variable must be specified with a type so that the interface generator can understand how it may be manipulated. For example, the Controls.Mode state in Figure 2 has

an enumerated type. We define seven primitive types that may be associated with a state variable:

- binary
- boolean
- enumerated
- fixed point
- floating point
- integer
- string

Many of these types have parameters that can be used to restrict the values of the state variable further. For example, the integer type can be specified with minimum, maximum, and increment parameters.

It is important to note that complex types often seen in programming languages, such as records, lists, and unions, are not allowed to be specified as the type of a state variable. Complex type structures are created using the group tree, as discussed below.

Label Information

The interface generator must also have information about how to label the interface components that represent state variables and commands. Providing this information is difficult because different form factors and interface modalities require different kinds of label information. An interface for a mobile web-enabled phone will probably require smaller labels than an interface for a PocketPC with a larger screen. A speech interface may also need phonetic mappings and audio recordings of each label for text-to-speech output. We have chosen to provide this information with a generic structure that we call the *label dictionary*.

Each dictionary contains a set of labels, most of which are plain text. The dictionary may also contain phonetic representations using the ARPAbet (the phoneme set used by CMUDICT [2]) and text-to-speech labels that may contain text using SABLE mark-up tags [15] and a URL to an audio recording of the text. The assumption underlying the label dictionary is that every label contained within, whether it is phonetic information or plain text, will have approximately the same meaning. Thus the interface generator can use any label within a label dictionary interchangeably. For example, this allows a graphical interface generator to use a longer, more precise label if there is lots of available screen space, but still have a reasonable label to use if space is tight. Figure 2 shows the label dictionary, represented by the `<labels>` element, for a number of states. The dictionary for the Controls group has two text labels and a text-to-speech label.

Group Tree

Interfaces are always more intuitive when similar elements are grouped close together and different elements are kept far apart. Without grouping information, the play button for the CD player might be placed next to the stop button for the Tape player, creating an unusable interface. We avoid this by explicitly specifying grouping information using a group tree.

We specify the group tree as an *n*-ary tree that has a state variable or command at every leaf node (see Figure 3).

```
<?xml version="1.0" encoding="utf-8"?>
<spec name="MediaPlayer" version="PUC/2.0">
  <labels>
    <label>Media Player</label>
  </labels>

  <groupings>
    <group name="Controls" is-a="media-controls">
      <labels>
        <label>Play Controls</label>
        <label>Play Mode</label>
        <text-to-speech text="Play Mode"
                        recording="playmode.au"/>
      </labels>

      <state name="Mode">
        <type>
          <enumerated>
            <item-count>3</item-count>
          </enumerated>
          <valueLabels>
            <map index="1">
              <label>Stop</label>
            </map>
            <map index="2">
              <label>Play</label>
            </map>
            <map index="3">
              <label>Pause</label>
            </map>
          </valueLabels>
        </type>

        <labels><label>Mode</label></labels>
      </state>

      <group name="TrackControls">
        <command name="PrevTrack">
          <labels><label>Prev</label></labels>

          <active-if>
            <greater-than state="PList.Selection">
              0
            </greater-than>
          </active-if>
        </command>

        <command name="NextTrack">
          <labels><label>Next</label></labels>

          <active-if>
            <less-than state="PList.Selection">
              <ref-value state="PList.Length"/>
            </less-than>
          </active-if>
        </command>
      </group>
    </group>

    <list-group name="PList">
      <state name="Title">
        <type><string/></type>
        <labels><label>Title</label></labels>
      </state>

      <state name="Duration" is-a="time-duration">
        <type><integer/></type>
        <labels><label>Duration</label></labels>
      </state>
    </list-group>
  </groupings>
</spec>
```

Figure 2. A sample specification for a media player with a few basic functions and a play list.

State variables and commands may be present at any level in the tree. Each branching node is a "group," and each group may contain any number of state variables, commands, and other groups. We encourage designers to make the group tree as deep as possible, in order to help space-constrained interface generators. These generators can use the extra detail in the group tree to decide how to split a small number of controls across two screens. Interface generators for larger screens can ignore the deeper branches in the group tree and put all of the controls on one panel.

Complex Types

Our specification language uses the group tree to specify complex type structures often seen in programming languages, such as records, lists, and unions. We chose this approach because we felt it simplified our language, and followed the principle of "one way to specify anything." If we had chosen to specify complex types within state variables, then authors could have specified related data either as a single variable with a record data type or as multiple variables within a group.

To support complex types, we have added several special group tags. Figure 2 shows an example of the `list-group` tag that we added for specifying lists. List groups have two implicit variables to track the length of the list and the current selection(s). State variables that are specified within the list group will have multiple values associated with them, one for each item in the list. Multi-dimensional lists can be created by nesting list groups. We have also developed a special dependency operator for lists that can be true if all items, any items, or no items in the list match a dependency equation. The `union-group` is similar to the `list-group`, but acts like a union from the C programming language.

Dependency Information

The two-way communication feature of the PUC allows it to know when a particular state variable or command is unavailable. This can make interfaces easier to use, because the components representing those elements can be disabled. The specification contains formulas (see the `<active-if>` element in Figure 2) that specify when a state or command will be disabled depending on the values of other state variables, currently specified with several types of dependencies: equal-to, greater-than, less-than, defined, and others. Each state or command may have multiple dependencies associated with it, combined with the logical operations AND and OR. These formulas can be processed by the PUC to determine whether a component should be enabled when the appliance state changes.

We have discovered that dependency information can also be useful for structuring graphical interfaces and for interpreting ambiguous or abbreviated phrases uttered to a speech interface. For example, dependency information can help the speech interfaces interpret phrases by eliminating all possibilities that are not currently available. The processing of these formulas is described elsewhere [7].

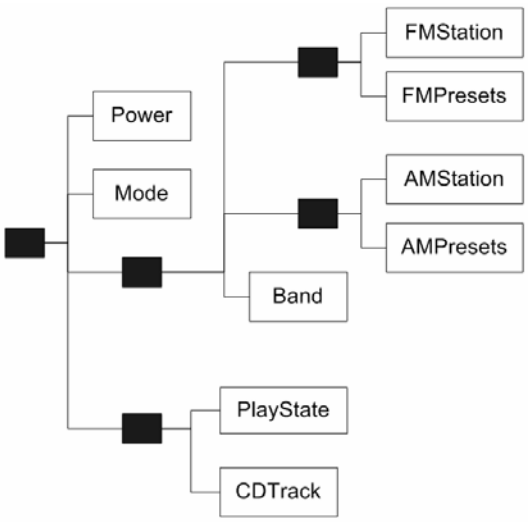


Figure 3. A sample group tree for a shelf stereo with both a CD player and radio tuner. The black boxes represents groups and the white boxes with text represent state variables. The mode variable indicates which source is being played through the speakers.

Smart Templates

A common problem for automatic interface generators has been that their interface designs do not conform to domain-specific design conventions that users are accustomed to. For example, an automated tool is unlikely to produce a standard telephone keypad layout. This problem is challenging for two reasons: the user interface conventions used by designers must be described, and the interface generators must be able to recognize where to apply the conventions through analysis of the interface specification. Some systems [19] have dealt with this problem by defining specific rules for each application that apply the appropriate design conventions. Other systems [5] rely on human designers to add design conventions to the interfaces after they are automatically generated. Neither of these solutions is acceptable for the PUC system. Defining specific rules for each appliance will not scale, and a PUC device cannot rely on user modifications because its user is not likely to be a trained interface designer. Even if the user was trained, he or she is unlikely to have the time or desire to modify each interface after it is generated, especially if the interface was generated in order to perform a specific task.

We have developed one solution to this problem called *Smart Templates* [9], where the PUC specification language's primitive type information is augmented with high-level semantic information. For example, the `media-controls` template defines that a state variable with particular attributes controls the playback of some media. Figure 2 shows how the `media-controls` Smart Template is indicated using the `is-a` attribute in our specification language. PUC interface generators can use the information added by a Smart Template to apply design conventions and make interfaces more usable. If an interface generator does not recognize a template however, a

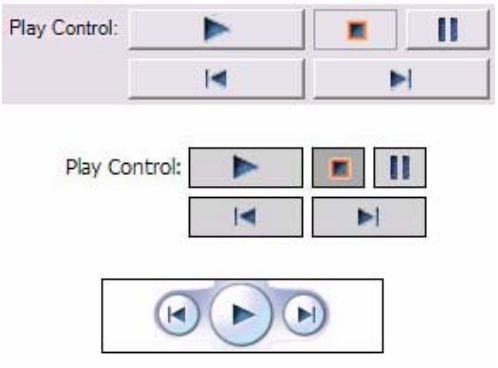


Figure 4. Media controls rendered for a Windows Media Player interface on each of our three platforms. At the top is the desktop, the middle is PocketPC, and the bottom shows Smartphone. The Smartphone control maintains consistency for the user by copying the layout for the Smartphone version of Windows Media Player, the only media player application we could find on that platform. This interface overloads pause and stop onto the play button.

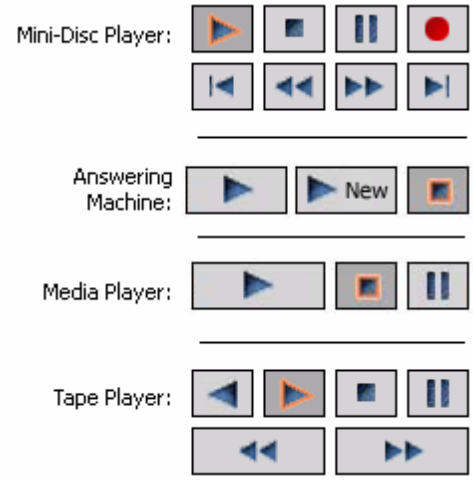


Figure 5. Different arrangements of media playback controls automatically generated for several different appliances from a single Smart Template (`media-controls`).

user interface can still be created because Smart Templates are constructed from the primitive elements of our specification language. Figure 4 shows the same instance of a Smart Template rendered on different platforms.

An important innovation is that Smart Templates are parameterized, which allows them to cover both the common and unique functions of an appliance. For example, the media playback template supports play and stop, but also optional related functions such as next track for CDs, fast-forward and reverse-play for tape players, and "play new" for phone answering machines (see Figure 5). Smart Templates also give appliances the flexibility to choose a functional representation that matches their internal implementation. For example, our `time-duration` Smart Template allows single state variables with integer or string types, or multiple state variables (e.g. a state for hours and another for minutes).

We have built a preliminary implementation of Smart Templates into the existing PUC system. So far the PUC supports a few Smart Templates: `media-controls`, `time-duration`, `image`, and `image-list`. We plan to implement many more, including `date`, `mute`, `power`, and `volume`. We expect that some Smart Templates will naturally combine with others to create new templates. For example, date and time are often used together, as are volume and mute. We hope to implement Smart Templates in such a way that templates can be flexibly combined with less work than creating a new template from scratch.

EVALUATION

We have not yet conducted any formal evaluation of our specification language, but we have used it to specify more than twenty appliances ranging from stereos and telephones to elevators and car navigation systems. We have used those specifications as the basis for generating graphical user interfaces on PocketPCs, Microsoft Smartphones, and desktop computers, and speech user interfaces using the Universal Speech Interfaces framework [14]. This section informally discusses some strengths and weaknesses that we have found with the language.

The main strengths of the language come from the design principles that we started with. Appliance descriptions are often a reasonable size, even for our largest and most complicated appliances. Our specification for an Audiophase 5CD shelf stereo system, which has more than 50 states, is 25KB. The specification for the GM Yukon Denali navigation system, which has more than 80 states, is 41KB. These sizes are perfectly reasonable for transmission and processing on the devices we are targeting.

Our language also seems to be reasonably easy to learn. Four undergraduate students have learned the language over the course of the project. Each student picked up the basics of the language in a day and was proficient within about two weeks. The most difficult aspects of writing an appliance specification are determining the appliance's variables and commands, and designing the group tree structure. We believe these aspects are difficult in general, and do not represent weaknesses in our design.

The main weakness of the language is the lack of any task information. For many appliances this is not a problem because all of the tasks have only one step. For example, "play the tape" or "increase the volume." With complex appliances, such as a car navigation system, this is not always the case, and the lack of task information may lead to lower quality generated interfaces for these appliances.

FUTURE WORK

We are planning to conduct a formal evaluation of the specification language and interface generators as part of the first author's thesis work. This will involve specifying more complex appliances and further testing the descriptiveness of the language.

We are planning a new feature of the PUC system that will generate a single user interface for multiple appliances that have been connected together. One example of a use for this feature is a typical home theater, which includes separate VCR, DVD player, television, and stereo appliances, but might be more easily thought of as a single integrated system. A PUC interface for a home theater would ideally have features like a "Play DVD" button that would turn on the appropriate appliances, set the TV and stereo to the appropriate inputs, and then tell the DVD player to "Play."

This feature will require some additions to our language to describe how appliances are connected together. Task information will also be required to support features like the "Play DVD" button, but the task information will be distributed among each of the different appliances. This is different from previous task languages [11] which have assumed that all task information is available in one location. Designing and building this distributed task language is a major area of future work for the PUC project.

CONCLUSION

We have discussed a language for describing appliances. The language is the basis for a system that automatically generates remote control user interfaces. We have used our specification language to describe more than twenty appliances from telephones to elevators to vehicle navigation systems. We have also written software that uses our language to automatically generate graphical user interfaces for handheld computers, mobile phone, and desktop computers, and speech interfaces using the Universal Speech Interfaces framework. We believe that the PUC specification language is at the appropriate level, and contains the right features to be successfully used for virtually all appliances and for many other tasks as well. We would welcome widespread adoption of the PUC specification language and collaboration with others.

ACKNOWLEDGMENTS

This work was conducted as a part of the Pebbles project, and the speech interface generator was implemented as a part of the Universal Speech Interfaces project. This work was funded in part by grants from NSF, Microsoft, General Motors, and the Pittsburgh Digital Greenhouse, and equipment grants from Mitsubishi Electric Research Laboratories, VividLogic, Lutron, and Lantronix. The National Science Foundation funded this work through a Graduate Research Fellowship for the first author and under Grant No. IIS-0117658. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation.

REFERENCES

1. Abrams, M., Phanouriou, C., Batongbacal, A.L., Williams, S.M., and Shuster, J.E. "UIML: An Appliance-Independent XML User Interface Language," in *The Eighth International World Wide Web Conference.* 1999. Toronto, Canada
2. CMU, "Carnegie Mellon Pronuncing Dictionary," 1998. http://www.speech.cs.cmu.edu/cgi-bin/cmudict.
3. Eustice, K.F., Lehman, T.J., Morales, A., Munson, M.C., Edlund, S., and Guillen, M., "A Universal Information Appliance." *IBM Systems Journal*, 1999. **38**(4): pp. 575-601.
4. Hodes, T.D., Katz, R.H., Servan-Schreiber, E., and Rowe, L. "Composable ad-hoc mobile services for universal interaction," in *Proceedings of the Third annual ACM/IEEE international Conference on Mobile computing and networking (ACM Mobicom'97).* 1997. Budapest Hungary: pp. 1-12.
5. Kim, W.C. and Foley, J.D. "Providing High-level Control and Expert Assistance in the User Interface Presentation Design," in *Proceedings INTERCHI'93: Human Factors in Computing Systems.* 1993. Amsterdam, The Netherlands: pp. 430-437.
6. Nichols, J., Myers, B.A. "Studying The Use Of Handhelds to Control Smart Appliances," in *23rd International Conference on Distributed Computing Systems Workshops (ICDCS '03).* 2003. Providence, RI: pp. 274-279.
7. Nichols, J., Myers, B.A., Higgins, M., Hughes, J., Harris, T.K., Rosenfeld, R., Pignol, M. "Generating Remote Control Interfaces for Complex Appliances," in *UIST 2002.* 2002. Paris, France: pp. 161-170.
8. Nichols, J., Myers, B.A., Higgins, M., Hughes, J., Harris, T.K., Rosenfeld, R., Shriver, S. "Requirements for Automatically Generating Multi-Modal Interfaces for Complex Appliances," in *ICMI.* 2002. Pittsburgh, PA:
9. Nichols, J., Myers, B.A., Litwack, K. "Improving Automatic Interface Generation with Smart Templates," in *Intelligent User Interfaces.* 2004. Funchal, Portugal: pp. 286-288.
10. Olsen Jr., D.R., Jefferies, S., Nielsen, T., Moyes, W., and Fredrickson, P. "Cross-modal Interaction using Xweb," in *Proceedings UIST'00: Symposium on User Interface Software and Technology.* San Diego, CA: pp. 191-200.
11. Paterno, F., Mancini, C., Meniconi, S. "ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models," in *INTERACT.* 1997. Sydney, Australia: pp. 362-269.
12. Ponnekanti, S.R., Lee, B., Fox, A., Hanrahan, P., and T.Winograd. "ICrafter: A service framework for ubiquitous computing environments," in *UBICOMP 2001.* 2001. Atlanta, Georgia: pp. 56-75.
13. Puerta, A., Eisenstein, J. "XIML: A Common Representation for Interaction Data," in *7th International Conference on Intelligent User Interfaces.* 2002. San Francisco: pp. 214-215.
14. Rosenfeld, R., Olsen, D., Rudnicky, A., "Universal Speech Interfaces." *interactions: New Visions of Human-Computer Interaction*, 2001. **VIII**(6): pp. 34-44.
15. Sproat, R., Hunt, A., Ostendorf, P., Taylor, P., Black, A., Lenzo, K., Edgington, M. "SABLE: A Standard for TTS Markup," in *International Conference on Spoken Language Processing.* 1998. Sydney, Australia:
16. Szekely, P. "Retrospective and Challenges for Model-Based Interface Development," in *2nd International Workshop on Computer-Aided Design of User Interfaces.* 1996. Namur: Namur University Press. pp. 1-27.
17. UPnP, "Universal Plug and Play Forum," 2003. http://www.upnp.org.
18. Vanderdonckt, J. "Knowledge-Based Systems for Automated User Interface Generation: the TRIDENT Experience," in *Technical Report RP-95-010.* 1995. Namur: Facultes Universitaires Notre-Dame de la Paix, Institut d' Informatique:
19. Wiecha, C., Bennett, W., Boies, S., Gould, J., and Greene, S., "ITS: A Tool for Rapidly Developing Interactive Applications." *ACM Transactions on Information Systems*, 1990. **8**(3): pp. 204-236.
20. Zimmermann, G., Vanderheiden, G., Gilman, A. "Prototype Implementations for a Universal Remote Console Specification," in *CHI'2002.* 2002. Minneapolis, MN: pp. 510-511.

Practical experiences with device independent authoring concepts

Oskari Koskimies
NRC, NOKIA GROUP
FIN-00045, Finland
oskari.koskimies@nokia.com

Michael Wasmund
IBM Deutschland Entwicklung GmbH
D-71003 Boeblingen, Germany
mwasmund@de.ibm.com

Peter Wolkerstorfer
Center for Usability Research & Engineering
A-1110 Vienna, Austria
wolkerstorfer@cure.at

Thomas Ziegert
SAP AG, Corporate Research
D-76131 Karlsruhe, Germany
thomas.ziegert@sap.com

ABSTRACT
The development of web applications for mobile and other non-desktop devices using established methods often requires a tremendous development effort. A major challenge therefore is to find sound approaches enabling the cost efficient application development for multiple devices of varying technical characteristics. Newer approaches are based on a device independent mark up language, which is then adapted to meet the special characteristics of the accessing device. This paper describes our approach to single authoring, which was developed in a large European research project[1]. The CONSENSUS project has developed a device-independent language profile based on XHTML 2.0 and XFORMS 1.0 and implemented a compliant rendering engine. In this paper we will describe our approach to enable single authoring and summarise the lessons we have learnt.

Keywords
Single authoring, Device Independence, Pagination, Adaptation

INTRODUCTION
The WWW has established itself as one of the most important sources for information and as a perfect infrastructure for applications, which need to be accessible from anywhere. Web-enabled devices potentially offer access to the globally adopted infrastructure. But currently, most web content is optimised for the usage on a PC. This is still true despite the efforts of the Web Accessibility Initiative [1] which covers the universal accessibility of web content. As more and more non-desktop devices enter the market, a convenient way to access web content and web applications using such devices is required. The industry addressed that requirement for a limited set of applications particularly in the B2E (Business-to-Employee) domain by developing device specific versions of those applications. Such an application-specific approach tends to be too costly and not manageable if scaled to a large number of diverse devices and applications. Therefore, more generic ways to prepare web content in a device-independent way are necessary.

Various approaches have been proposed to address this challenge. Some of them (e.g. [2, 3]) automatically compile web content to fit on a target device. These approaches are based on HTML and deploy heuristics in addition to tag information to extract structure information, caused by the fact that HTML does not provide the necessary semantics needed to perform the conversion to other mark up languages. Therefore, other approaches replace HTML by another, semantically richer mark up language to serve as a basis for adaptation [4, 5, 6]. However, none of these proposals is standards-based and they were therefore not widely adopted in the market. To open the path for a widely adopted device-independent mark up language for authoring web content, two considerable efforts have been launched: a) The W3C chartered the Device Independence Group to establish specifications supporting single authoring, b) a consortium of six European companies, named CONSENSUS [7], has built a prototype which implements authoring and conversion tools for a Renderer-Independent Markup Language (RIML), specified by this consortium as well. Both efforts cooperate intensively. The ultimate vision of device independence as stated in the charter of the W3C Device Independence working group [8] is to provide “Access to a Unified Web from Any Device in Any Context by Anyone”.

REQUIREMENTS FOR DEVICE INDEPENCE
The requirement to feed a multitude of web browsers with content from the web in a uniform way is not new. The W3C recognised this requirement and evolved HTML not only to meet XML compliance (resulting into XHTML 1.0 and XHTML 1.1), but also catered for a better separation of content from presentational aspects.

[1] IST-Programme / KA4 / AL: IST-2001-4.3.2. The project CONSENSUS is supported by the European Community. This document does not represent the opinion of the European Community. It is also the sole responsibility of the author and not the responsibility of the European Community using any data that might appear therein.

Currently, there are three more prominent examples of how the newest XHTML specification, coined XHTML 2.0 Working Draft [9], addresses the above requirement.

First of all, the W3C HTML Working Group did the bold step to remove all input-related mark up, such as the well-known HTML input element, completely. Instead, all user interaction is now delegated to a separate specification called XFORMS 1.0 [10]. Taking a closer look on XFORMS, one recognises it attains a new level of abstraction. An author now expresses the intent, for example, that a user is expected to select one out of many items, but not the exact presentation of the list of items anymore. The select element may be presented as radio buttons or a drop down box or in any other way at the discretion of the browser which interprets the mark up. Other relevant features of XFORMS are described elsewhere [10].

A second feature of XHTML 2.0 which underlines the separation of presentation from content is the new way to express content structure. The content is expected to be structured by the author as a consecutive nested list of sections. Each section may bear a heading, however, headings do not have an explicit weight anymore, but there is only a general h element left. The weight of a heading results from the nesting level of the section it is associated to. XHTML 2.0 leaves it to the interpreting browser to present the heading in a proper format.

Finally, XHTML 2.0 deprecates the use of formatting directives such as displaying text in bold or italic in favour of using Cascading Stylesheets (CSS) [14]. The use of CSS is a further development which supports development of device-independent content as we will see.

Despite the advances of the XHTML specification outlined above, the Consensus project identified shortcomings which prevent the use of XHTML 2.0 as the only means to express content in a device-independent way. Several other projects [18, 19, 20] also tried to fill these gaps by adding proprietary mark up, whereas the Consensus project strives for a standardised way to close the gap.

The shortcomings which we identified can be attributed to the following areas:

Pagination, which is also called decomposition, is a quite familiar concept in the printing domain. Pagination occurs, when the document's content does not fit on a single page/screen. If that case happens, the content is split into two or more chunks, each of them fitting on a single page/screen. For non-interactive content i.e. printed pages, the content will be split when the page is full and page numbers are inserted to provide a basic means of navigation. Specifications such as XML Print [17], CSS3 Paged Media [13], and XSL-FO [12] address that.

For interactive content, decomposition techniques known from printing are not sufficient for several reasons: a) generated navigation in form of page numbers does not provide good usability when exploring highly structured content such as output from a portal page. b) The pagination of forms carries additional usability issues as everyone who filled multiple consecutive forms in a web application can imagine. c) Tables need particular attention, since the distribution of content organised in tables across a sequence of screens may not reflect the intent of the author anymore or can be just inappropriate for display capabilities of certain devices.

Navigation. As indicated above, Pagination requires means to navigate between the pieces the original content was split into. Navigation links should be generated automatically during pagination, but authors may want to influence placement and appearance of the additional navigation. The application may already generate a similarly structured output, such as a partial list of search results. Application-generated navigation can interfere with the additional navigation generated by pagination.

Layout. For PC-based browsers, a sophisticated layout, based on deeply nested HTML-tables, may be suitable for proper presentation of content. However, for most of the mobile devices, a much simpler layout needs to be applied due to the limited screen space. To support a meaningful pagination of content it is furthermore desirable to be able to specify layout areas, to which content is associated dynamically. Some of these areas i.e. their content might be repeated on each page or screen, similar to running headers or footers, to allow for navigation bars, screen titles, or copyright notices.

Voice. Finally, although XHTML 2.0 remarkably provides already guidance on how a table could be presented aurally, there is not much more support to present XHTML as voice. To compensate for that, the XHTML + Voice specification [16] was proposed, which basically allows for the insertion of VoiceXML into XHTML. However this is rather integration on a technical level and does not solve the problem of device-independence with respect to voice as such.

Currently, XHTML 2.0, despite its recent advances, does not really address the above list of problems. The language profile RIML, defined by the Consensus project [7], claims to provide a solution for these problem domains. The claim is supported by an implementation of a compliant rendering engine and associated field trial carried out with the help of usability experts.

Renderer Independent Markup Language

The RIML specification addresses the problem areas outlined above as follows:

Layout

To support a dynamic arrangement of content into varying layouts, RIML explicitly defines a layout element in the head of a document rather than mixing content and layout in the body. The layout consists of a hierarchy of containers, where content can be placed. Content is bound to layout via named sections. This is a similar concept as in XFORMS,

where the “content” of a form (called “model”) is placed in the head, and bound via reference to the presentational part of a form in the body.

RIML specifies a very simple set of layout containers, which are a) row, b) column, and c) frame as the innermost container. Beyond row and column, RIML provides a third type of container which is named “grid”. The grid delegates the actual layout to the interpreting browser or adaptation process by just expressing a preferred layout (for example, columns), but allowing the interpreter to arrange the content differently if suitable on the target device.

Layout containers in RIML can be paginating or nonpaginating. Paginating containers allow their content to be distributed on consecutive screens, whereas the non-paginating containers repeat their content on every screen (i.e., header, footer).

With help of these simple layout concepts, we were able to arrange content for a wide variety of devices in meaningful way.

Figure 1 and Figure 2 in the sequel of this documents show example layout definitions in RIML mark up and the resulting output of the respective browsers, when the mark up has been processed by the adaptation engine. The Communicator was emulated by a common PC-browser here. Furthermore both examples show the use of pagination directives.

Pagination

Similarly to a written document, where an author can enforce a page break or leave the determination of best page break to the interpreter, the project discussed intensively whether such directives should be provided by the author. The outcome of the discussion, accompanied by several experiments, was that RIML does not provide explicit pagination directives for the author similarly to a “begin-new-page-here” directive, because such directives do not support device-independent authoring.

Instead of this, we defined a predictable behaviour of RIML mark up interpreters when tables or sections are encountered, which do not fit on a single screen. Examples of such rules are: a) keep those parts of a table together which are enclosed by a `tbody` element, b) avoid splitting sections. If these example rules cannot be applied because the resulting amount of content is still too large, further decomposition rules apply. The important point here is to delegate the “best” method to paginate to the interpreting system rather than allowing the author to “hardcode” such directives. The content of the menu and navigation frames (menu-frame, nav-frame) appears on every paginated screen, whereas the content of the content frame gets split.

Navigation

Opposite to pagination, our studies and practical experiments revealed that authors need means to control the type and appearance of so-called generated navigation. That is navigation which is optionally inserted into the presented output when pagination occurs. Authors want to be able to determine: a) where on the screen the navigation is displayed (top, bottom, sidebar), b) how it is displayed (text or image), c) how much of it is displayed (just “next” and previous” or all related links). The latter directive is particular important in case of deeply structured content such as a portal page, where a user may navigate from portlet to portlet (assuming these are distributed across several screens), but also within a portlet, which content was paginated. RIML therefore introduced a navigation directive called “scope”.

An example of how to use navigation directives can be found at the bottom of Figure 2.

Voice

One of the most challenging topics of the project was voice support. Other than XHTML+Voice [24], we aimed for a more abstract way of voice integration by interpreting existing XHTML mark up in a suitable way to produce voice output. However, voice is extremely different from visual presentation and input, so that we had to add some extra support for voice interaction. Experiments revealed that using RIML’s layout, pagination, and generated navigation concepts for voice interaction was not really feasible. Theoretically, it may be possible to generate a menu, providing a voice dialogue stating the available choices from the hierarchy of nested sections. However, practical experiments revealed, that the usability of such an approach would be unsatisfactory. Instead, the `nl` (navigation list) element of XHTML2 is used to specify such menus, so that the voice system can tell the user the available choices. For graphical browsers these menus are rendered as a table or a list of menu options. This matches the original XHTML2 semantics [10], even though default behaviour is slightly different (no folding).

Additionally, a voice-specific dialog element (global menu) in the document header can be used to define voice navigation options that are always available, e.g. so that the user can start a new search by saying “search” at any time. Therefore, for voice interaction, RIML’s layout and pagination directives are ignored, and the author must provide a navigation list and possibly a dialog element instead.

Content Selection

Ideally, an author would produce documents, which fit all available user devices thanks to the power of RIML and intelligent adaptation mechanisms. However, there is always the issue of how to handle the control over the final presentation. Usually authors want to influence the final result on the device and therefore need a way to maintain control of the appearance if they wish.

Beyond that, authors may want to provide different resources for different classes of devices. Such resources can be for example a long text versus a short text, or different versions of a corporate logo. To support such

requirements, a number of languages already provide so-called content selection [11, 15]. For RIML we adopted and extended the content selection as defined by SMIL.

The application of content selection allows the author to conditionally include certain content. The conditions can be specified as a function of device properties such as the width of the screen.

Field trial results

The distinguishing factor of the Consensus project compared to many other attempts in the problem domain was the inclusion of usability experts from the very beginning and practical experiments up to real field trials with existing applications ported to RIML. That allowed us to assess whether our specification (and also the matching implementation) is practical. One of the main goals of Consensus was the development of usable applications. Therefore it is crucial that the user interfaces of applications coming from the adaptation engine produce a sufficient user experience. In order to evaluate whether RIML reached this goal we have done a field trial. The field trial tests were performed at the connection speeds provided by GSM, GSM HCSD, and GPRS. This approach was chosen in order to create a test situation that is as realistic as possible, e.g. to take response times into account. Audio and video signals from all tests were recorded. User actions on the devices were filmed. The problems that occurred can be summarised in the following four categories:

Fold problems: users did not scroll beyond empty lines on devices with small screens so they missed content; in addition to that a kind of "vertical fold problem" occurred: when a navigation bar is not considered for the usage on narrow screens by the developer the users have to scroll vertically to get all navigational functions which has not been recognized by the users – so they missed important navigation.

Mandatory input fields: text fields used to enter dates or time where rather cumbersome for users who used a mobile phone due to the input modality. This problem resulted mainly from the fact, that the implementation was limiting the use of optimal input controls for certain data types (see also the lessons learnt section).

Layout problems: the automatic adaption of the layout has not always resulted in an optimal use of the screen real estate, which was mainly caused by the limited pagination algorithm of the implementation.

Overall Results

During the tests following two main questions have been answered:

Is the development of mobile applications with RIML supported by the Consensus tools faster than the development with native (already avail-able) tools?

Is it possible to develop usable applications that meet high quality standards with RIML and the Consensus tools?

The answer to both questions is YES, if an iterative design process is applied (prototyping, test, implementation, test, final implementation) and if the developers consider all the restrictions of the targeted devices already at the beginning of a development project. In particular this result is promising because RIML and the Consensus tools are only prototypes, because the developers had rather little time to become familiar with RIML and with the tools and because the applications had to be developed under tight time constrains.

The overall goal of the project was to find ways to lower the costs for providing web content to a multitude of primarily mobile devices while maintaining a high degree of usability. The field trial basically confirmed that our approach fulfils the promise. The field trial also provided us with a wealth of "lessons learned", which are shared below.

Lessons learnt

Content control

The SMIL-based content control attributes provided with RIML proved to be especially useful for controlling layout. A typical application of content control is that an author specifies a layout for mini-screen devices, portrait-screen devices, PDAs, and PCs. A perfect browser or adaptation system may not need such author provided layout alternatives. However, today's adaptation mechanisms, including those provided by the Consensus project, cannot derive a highly usable presentation from one layout only. The W3C Device Independent Group is working on a specification for Content Selection which is similar to RIML, but includes XPATH-based notation expressing the conditions. This underlines our finding that Content Selection is a necessary ingredient for device-independent authoring.

Pagination

It is probably not surprising that pagination was found to be a useful feature, both for forms and for general content. Pagination of tables is worth to be mentioned especially, however. Static content can be paginated relatively easily, but application-generated data is harder – the pagination algorithm would then have to be built in to the application. Such application-generated data frequently takes the form of a table, so being able to automatically paginate tables (and retain column headers on subsequent pages) is essential.

Layout

The separation of content and layout in RIML proved to be effective. For voice interaction, the layout is ignored and only the content is "read to the user". Since layout is more related to styling than to content, there is currently a discussion in the Device Independence group of the W3C whether layout should be rather expressed by a set of new CSS directives instead of adding XML mark up to XHTML.

In order to effectively separate content and layout, both should be loosely coupled. This also aids the cooperation of content author and layout designer. While the method of referring content to layout regions was sufficient for our example applications, in hindsight this type of coupling is too tight and a more flexible one should be employed. An intriguing possibility would be to use separate bind elements that refer to content using XPath expressions, similar to those in XForms.

We were not able to test the promising concept of "grid" within the course of the project anymore - a layout container which expresses a preference for arrangement of content, but leaves the eventual arrangement to the interpreter (adaptation engine or browser). For example, content could be arranged vertically for portrait type screens and horizontally for landscape screens. In many cases a grid layout would make it unnecessary to provide different layouts for different screen types.

XForms Event Support

In order to support as many different devices as possible, one cannot require too much from the client devices regarding the supported mark up. In particular it cannot be assumed that scripting is always available, since scripting support considerably increases the browser's footprint. This limits the possibilities for XForms support when using a server-side XForms processor. While the server-side approach is still feasible, some XForms features cannot be fully supported, especially events. Even for those features that can be implemented on server side, the long latency in mobile networks would severely reduce usability. Consider the classic credit card example, where input fields for credit card information are dynamic and appear only once credit card has been chosen as the method of payment. When using a server-side XForms processor and a GPRS data connection, there would be a very noticeable delay between selecting the credit card option and the appearance of the fields for credit card information.

Nevertheless, in field trials authors lamented the lack of event support. They would have liked to be able to e.g. instantly react when the user selects a particular choice from a drop-down list, which is the common case when using HTML forms. In general, any advanced user interface functionality requires some sort of event support.

Strong typing

The usability of XForms in an environment with a multitude of client device types depends to a large extent on how well implementations support different data types, especially for voice interfaces. For example, consider the date data type. As long as validation is done correctly, a fully conformant XForms client can still use the standard string input field for entering a date, requiring the user even to enter the separator characters himself. If only this minimal support for date input is provided, using a phone keypad to enter the date will result in a poor user experience. A better implementation will let the user enter the day, month and year into separate fields, and really good one will provide a specialized calendar tool for selecting a date.

Good input optimization based on data types is crucial for mobile devices. Not only the input facilities are more limited (such as a phone keypad), but they are frequently used in situations (e.g. while walking) where the user cannot spare his full attention for the application. These usage scenarios are particularly unforgiving of suboptimal user interfaces.

Voice interfaces are a special case. Current state-of-the-art speaker-independent speech recognition systems aimed for the use over telephone lines cannot interpret arbitrary speech; instead, the system has to know the possible input alternatives in advance. The set of possible input alternatives and their semantics is called a grammar (loosely defined). A grammar must be provided for each and every input control in a form, if a voice interface is to be used. However, if the data type for the input control is known, a grammar can be provided by the system automatically. For example, if the data type is date, a grammar might include words such as "January", "tomorrow", "first", etc.

Processing chain

Modelling the RIML adaptation engine as a chain of processors (a common way of modelling XML processing) allowed us to separate concerns like content selection, pagination and mark up mapping. This separation kept the complexity at a manageable level. However, in order to paginate XForms, pagination and XForms processing had to be integrated to some degree, and this resulted in one of the most complex modules of the prototype implementation.

Advanced pagination algorithms

As mentioned earlier, pagination is very useful. However, while conceptually simple, there are severe difficulties in implementing a good pagination algorithm when a range of different mark ups can be produced. Essentially, the pagination algorithm needs to know how much screen real estate a certain string of RIML content will take, once it has been converted to target mark up. This is obviously a highly non-trivial calculation, and the exact answer depends actually on the browser implementation of the client. The Consensus prototype implementation provided a very simple algorithm that essentially counted characters. Even this simplified pagination provided useful results, but sometimes fine-tuning of content was necessary to make the pagination work right. A more advanced algorithm that would have taken the properties of different mark up languages and tags better into account would have removed the need for fine-tuning. As such an algorithm would essentially be an expert system for the rendering behaviour of different mark up languages, it was not possible to develop it in the scope of the Consensus project.

Input data issues

A server-side XForms processor converts XForms to forms in the target mark up, such as HTML or WML (outbound data). It also converts name-value pairs in the submitted form data (inbound data) to XML, which is then sent to the application. Mark up mapping is the process that does the aforementioned conversions. A mark up mapper should be able to process both outbound and inbound data. In Consensus only the former was possible, and a standard form data submission protocol had to be followed by all mappers. This meant, among other things, that each instance data item had to map to exactly one form control in the target mark up (as the submission protocol implicitly assumed this). This limitation, however, made it impossible to do a good date input. The instance data had one item of type date (as it should). By default it is mapped to a single, free-form text input field, which is not exactly good usability. Optimally, this would have been mapped to three input controls, one for day, month and year each, possibly using selection lists. However, this would only have been possible with a mapper that would also have been able to convert the inbound data, mapping the three form data items to one instance data item. Another possibility would be a calendar tool, but it would have to be implemented wholly on server side (without scripting).

Voice

Our experiences in the trial were that voice browsers were not standardized enough yet to make it possible to provide a single VoiceXML mapping. Rather, separate mappings were provided for different voice browsers. Furthermore, mapping generic XForms forms to voice is not possible as the technology does not yet support free-form input (i.e. the basic input field without any typing) with speaker-independent voice recognition. The current design requires the inclusion of grammars for each form. However, when a data type is known, a grammar could often be automatically provided for it, as we noted earlier. In some cases though, the application must always provide the grammar. For example, in a phonebook application the grammar for the name input field is the set of names in the phonebook, which obviously must be provided by the application.

Conclusions

The Consensus project addressed the problem of content authoring for a large range of devices in a comprehensive way. We defined not only a language profile based on newest standards, but also usability guidelines which were partially implemented in a matching adaptation engine. The project concluded with a field trial in which we tested the language profile and its implementation with a real corporate application and real employees, proving the feasibility of the approach. The field trials showed that applications developed with RIML can be very usable with a range of devices. However, achieving good usability required an iterative design process (prototype, test, redesign, test, final design) just as it would for any application. RIML is not a silver bullet for developing applications with a good usability in one shot; rather it reduces the burden for achieving good usability over a range of devices by solving many well-known usability problems on the language/platform level.

In summary, the single-source authoring approach is feasible at least for data-driven applications such as enterprise web applications, and it clearly shows promise for reducing the cost of producing device-independent applications.

REFERENCES

1. Web Accessibility Initiative, http://www.w3.org/WAI/
2. Bickmore, T.W.: Digestor: Device-independent Access to the World Wide Web, Proceedings of 6th International WWW Conference (1997)
3. Schilit, B.N., Trevor, J., Hilbert, D., Koh, T.K.: m-Links: An Infrastructure for Very Small Internet Devices. Proceedings of the 7th Annual International Conference on Mobile Computing and Networking. Rome, Italy, (2001) 122-131
4. Puerta, A., Eisenstein, J.: XIML: A Common Representation for Interaction Data, www.ximl.org/documents/XIMLBasicPaperES.pdf
5. User Interface Markup Language, http://www.uiml.org
6. Eisenstein, J. et al.: Applying Model-Based Techniques to the Development of UIs for Mobile Computers, Proc. of the Conf. on Intelligent User Interfaces, Santa Fe, NM, USA, (2001)
7. Consensus Project Website, http://www.consensus-online.org
8. W3C's Device Independence Working Group, http://www.w3.org/2001/di/Group/
9. McCarron, S., et al (eds): XHTML2, W3C Working Draft 5 August 2002, work in progress, http://www.w3.org/TR/xhtml2/
10. Dubinko, M., Klotz, L. L., Merrick, R., Raman, T. V.: XForms 1.0, W3C Recommendation 14. October 2003, http://www.w3.org/MarkUp/Forms/
11. Hoschka, P. (eds): Synchronized Multimedia Integration Language (SMIL) 1.0 Specification, http://www.w3.org/TR/1998/REC-smil-19980615 (1998)
12. Adler, S. et al: Extensible Stylesheet Language (XSL), Version 1.0, http://www.w3.org/TR/xsl/
13. Lie, H. W., Bigelow, J. (eds): CSS3 Paged Media Module, work in progress,http://www.w3.org/TR/css3-page/
14. W3C Cascading Style Sheets, www.w3.org/Style/CSS/
15. CSS Media Queries, http://www.w3.org/TR/2002/CR-css3-mediaqueries-20020708/
16. XHTML + Voice Profile 1.0, http://www.w3.org/TR/xhtml+voice/
17. Bigelow, J., XHTML-Print, http://www.w3.org/TR/xhtml-print
18. Hirose, S., Kondo, G., "Application of Dharma, a Framework for Web Applications for Pervasive Terminals, to HTML Transforming Proxy Servers", IPSJ 59th Annual Convention (1999).
19. Lawrence Bergman, PIMA: A Model for Developing Pervasive Applications, IBM TJ Watson Research Center
20. Banavar, G. et al: An Application Model for Pervasive Computing, Proceedings of the Mobicom 2000

Figures

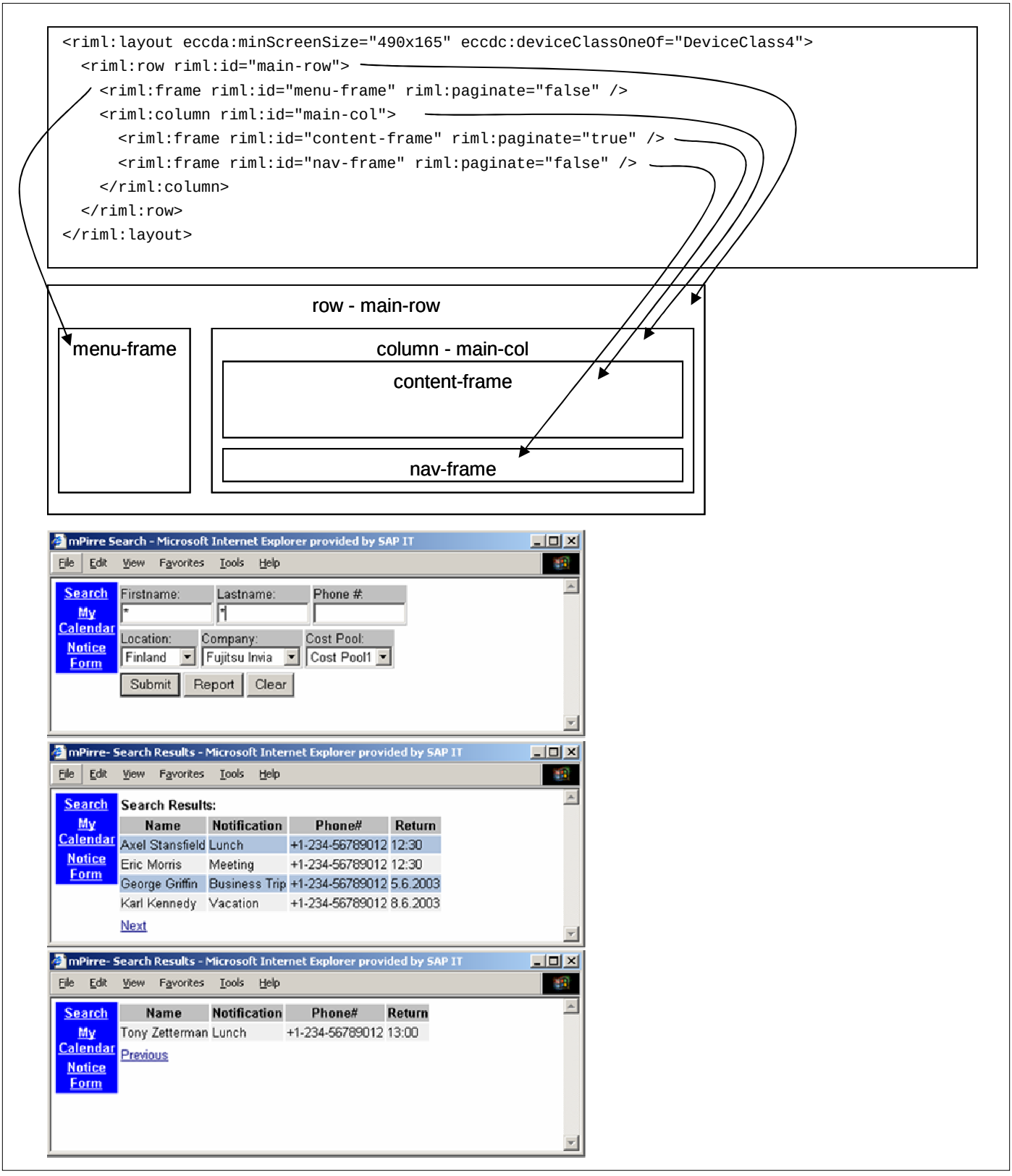


Figure 1: Layout and Pagination Example - Device Class 4 - Communicator-like Device

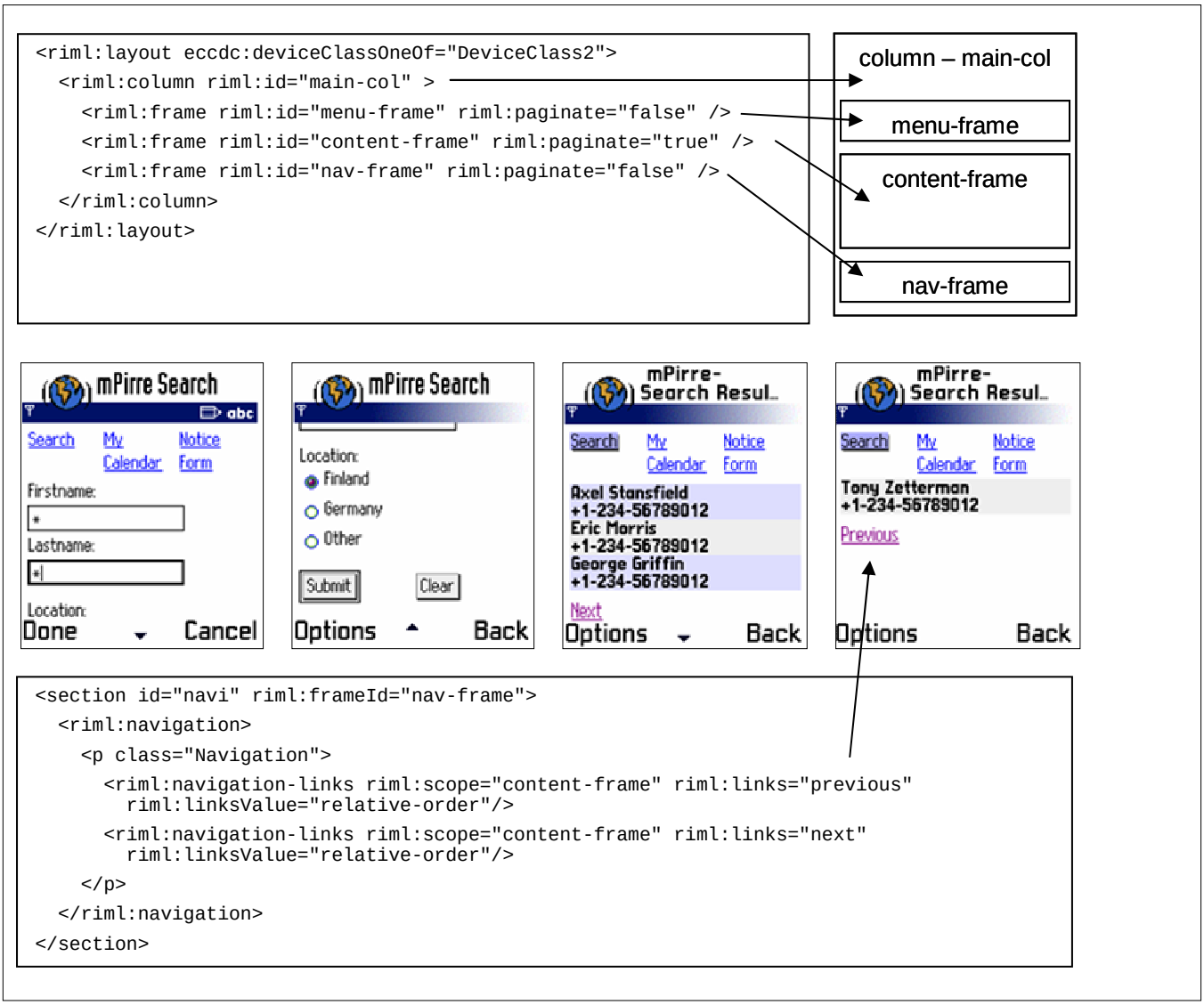


Figure 2: Layout and Pagination Example - Device Class 2 - Series 60-like Device

Dynamically generated multi-modal application interfaces - position paper -

Stefan Kost
TU Dresden, HTWK Leipzig
st_kost@gmx.de

ABSTRACT
This work approaches dynamic multi-modal application interfaces from a new point of view. The ongoing diversification of the user base and technology lays the foundation for the need of an holistic adaption infrastructure. Only designing individual adaption methods is not sufficient anymore. Providing such an infrastructure along with an open reference implementation is the objective of the Generalized Interface ToolKit (GITK) project. The software can generate, adapt and exchange interfaces at runtime. It works on various platforms and comes with several interface renderers. The solution is based on XML technology and defines an own markup language called Generalized Interface Markup Language (GIML).

Author Keywords
adaptive systems, adaptable systems, dynamic multi-modal application interfaces, UIMS

INTRODUCTION

In the last years a new development took place in our world. This work refers to it as *diversification*. It shows as two separate effects:

- **"technification" of all areas in life**
  Humans are outnumbered by technical devices already or real soon! Already these days it is nearly unavoidable to get in touch with technology. Therefore technology must be made accessible to everyone, everywhere and at every time not just physically.
- **"computerization" of devices**
  Many technical devices are more often multi-purpose appliance like computers. They have facilities for interaction and share basic common tasks.

So the challenge is to enable all the technology to all the people. This leads to a multi-dimensional adaption problem. The presumption that can be made here is, that software needs to adapt much more than it already does. Adaptation needs to work in a media-neutral fashion. It should be understood as a continuous process and not as something that happens once.

Interaction

This work deals with interfaces (see section Interfaces later in this article). Interfaces exist for the purpose of allowing *interaction*.

DEFINITION 0.1 (INTERACTION). *Interaction is the process of two or more systems exchanging data to perform a task.*

Participants of an interaction are called *interaction partners* or *interactors*. An *interactor* is a system with a variety of input and output channels. Each of these channels can submit data or stimuli of a certain type (like e.g. sound , visuals or touch). These submissions are subject to interpretation by the receiving system. The sum of the bandwidth

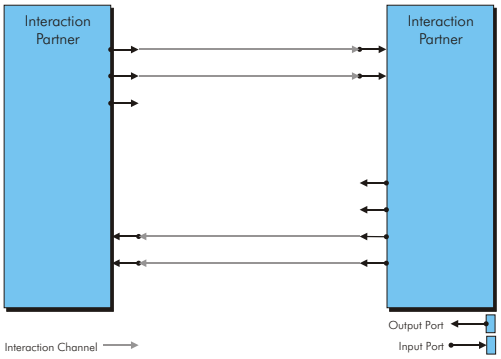


Figure 1: undisturbed interaction scenario

of the interaction channels defines the *potential interaction capabilities* of the system. These potential capabilities can only be used in the optimal case, where no obstacles hinder the interaction. The environment has such a blocking effect. Therefore the *effective interaction capabilities* are what remains after the varying blocking effect of the environment has been taken into account. [Stary, 1996, Dix et al., 1997]
It is obvious that the chances for successfully establishing enough links for an efficient interaction are not very good in the case shown in figure 2.
This work focuses on human-computer interaction and computer-computer interaction. To assure effective interaction, adaption is needed. For human-computer interaction it is desirable that the computer adapts to the needs of the human interaction partner. Using adaptive interfaces in the fields of

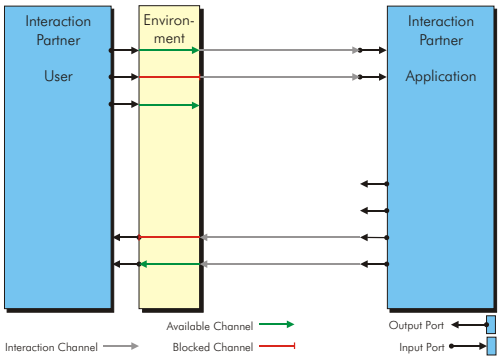


Figure 2: interaction scenario with environmental influence

computer-computer interaction allows to easily reuse an application in a different environment.

Adaption

Adaption is a key concept in this work, but also in the real world. Therefore a definition for the context of this work is required.

DEFINITION 0.2 (ADAPTION). *Adaption is the process of changing an object so that it complies to given requirements.*

One conclusion from this definition is that the object needs to be adaptable at all. It needs to offer different modes of operation that can be matched with the requirements. There are two kinds of adaption:

- **passive adaption** or **adaptable system** = system will be manually adapted by an external entity
- **active adaption** or **adaptive system** = system adapts itself automatically

[Fink et al., 1996]
Adapting an object needs knowledge about what changes are necessary for a desired effect. The overall knowledge regarding to adaption can be broken down on the base of single aspects.

DEFINITION 0.3 (ADAPTION METHOD). *An adaption regarding to one single aspect of the adaption object is the application of an adaption method. The method describes which changes are needed for specific requirements.*

The design of a good adaption method for human users requires knowledge from fields like cognitive science and psychology.
As presumed earlier in this article software needs to adapt. More precisely the interfaces are the *objects* to be adapted. The adaption process is controlled by parameters, the *adaption requirements*. In the case of this work these requirements are *adaption profiles* and consist of *environment profiles* and *user profiles*.

DEFINITION 0.4 (USER PROFILE). *The user profile describes the adaption requirements of the user and consists of the following parts:*

- *the **interaction capabilities** of the user as a communication partner. Interaction capabilities are a compound of the sensorical and motorical capabilities.*
- *the **interests and preferences** of the user relating to the style of the interaction.*
- *the **knowledge and competence** of the user regarding to the task.*

Defining an individual user profile is called *user modeling*. [Fink et al., 1997]

DEFINITION 0.5 (ENVIRONMENT PROFILE). *The environment profile describes a filter that applies to the capabilities part of the user profile. It temporarily restricts or even blocks certain interaction capabilities of the user.*

While the user profile can be seen as an object with nearly static properties, the environmental profile needs to be considered as highly dynamic.
To adapt an interface usually multiple adaption methods need to be applied. It sounds sensible to define an *adaption infrastructure* that handles the application of adaption methods. Designing such an infrastructure requires engineering skills from the area of computer science. Therefore the separation into adaption methods and adaption infrastructure reflects the relation to different areas in science.
Figure 3 graphically shows the relation of the previously defined terms for the scenario of human-computer interaction. Finally a short summary can be given:

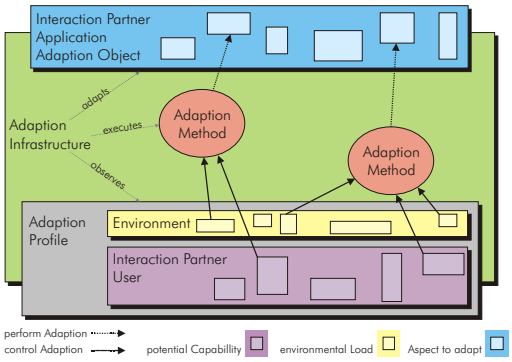


Figure 3: schematics of adaption and involved components

- the users' potential capabilities plus the current environment form the adaption profile
- the adaption profile controls the adaption process
- the adaption infrastructure provides means to read the profile and to choose and execute the respective adaption method
- the adaption method performs the adaption of one aspect of the application interface according to the requirements given by the adaption profile

[Stary, 1996, Dix et al., 1997]

Interfaces

DEFINITION 0.6 (INTERFACE). *An interface provides well defined access to functionality of an object from outside. It appears as a layer between two parties and aids their interaction.*

In [Phanouriou, 2000b] an interface is separated into four aspects:

- **structure**: the organization of interface objects
- **content**: resources used in the interfaces such as label texts and shortcut metaphors
- **style**: the presentation of interface objects
- **behavior**: defines the action to be performed on interaction with the interface objects

The application needs to provide structure, behavior and content. The presentation and choice of the content (e.g. for i18n) is dependent on the modality of the interface and the user profile. Therefore these aspects will be chosen by the adaption infrastructure. Finally style is an aspect that should be provided and handled by the adaption infrastructure.

Interface models

In the past various models for decoupled interface architectures have been suggested. A general criticism on models such as Seeheim, Arch, MVC and PAC is, that they aim to model adaptive systems, but do not represent the adaption process as such. These models only decouple components, but lack a definition of how adaption is driven (how to couple the right component-instances at run-time). Seeheim and Arch span a series of components between user and application, neglecting that there is an environmental influence affecting the interaction and that a user might carry out several tasks synchronously. In the past, when the models such as Seeheim and Arch have been defined, these two effects were hard to take into account for technical reasons or were less important. This has changed in the present. [Pfaff, 1985, various contributors, 1992]

Existing approaches

The goals of this work are similar to those of other projects. A big share of them either became dormant (AUIML) or seemingly have been discontinued (XIML). Another group of projects focuses on adaptive hypermedia applications. These projects usually develop adaption methods for their purpose and then an infrastructure to drive them. Furthermore there are solutions such a UIML and XUL which are active. XUL focuses on graphical interfaces only. It mainly serves as a operation system portability layer. Interfaces generated by UIML and XUL can not change their modality at runtime. With UIML the developer needs to specify all target interface variants that should be available later. Both languages use XML as a ”input file-format”. [Phanouriou, 2000a, Hyatt, 2000]. Finally some projects are quite similar like W3C XForms, but started in parallel or later as this work. XForms maintains separate XML documents for content and interface [Dubinko et al., 2003]. All approaches mentioned above have in common that they do not aim to provide a system, where interfaces can be adapted or even exchanged at runtime.

Aim of this the GITK project

The previous sections motivated that it is useful to distinguish between the *adaption infrastructure* and *adaption methods*. They further showed that an adaption has a multi-dimensional nature. Therefore a holistic approach to adaption is needed. Applications using this technology would then be adaptable, as they will use a pure functional interface description as an input and leave the generation of a concrete interface to the system.
In parallel more research is required in user modeling to define rich user profiles that can control the adaption process. This would turn the adaptable systems into adaptive systems. This work focuses on providing a fundamental adaption infrastructure, with a strong decoupling of application logic and interface presentation. On top of that a limited number of adaption methods will be implemented as a proof-of-concept. However it is not the objective of this project to develop new adaption methods, nor evaluating them.

THESIS

This work will show, that:

THESIS 0.1. *By limiting the presentational complexity a much greater universality can be achieved.*

THESIS 0.2. *There is no reason for adaptive solutions to mainly concentrate on graphical presentation.*

THESIS 0.3. *It is possible and even desirable to separate style related description from functional interface description.*

THESIS 0.4. *An interface can be generated, without the application needing to provide adaption profiles for different targets. In other words: even adaption methods can be generalized.*

THESIS 0.5. *It is possible and preferable to always have a default behavior, that can be overridden by adaption, instead of only relying on the adaption.*

THESIS 0.6. *A solution can be based on many standardized and well established technologies. In fact it can glue many specific solutions together, which already exist.*

THESIS 0.7. *Beside humans an application can be an end-user as well and therewith benefit from an adaptive solution.*

THESIS 0.8. *Adaptive technology is necessary for everyone and not just for minorities (like elderly or disabled people).*

SOLUTION

The solution presented in this work is called Generalized Interface ToolKit (GITK) and consists of three parts:

- an architecture related to the arch model that fits with the formerly defined adaption structure

- a domain independent markup-language that is called Generalized Interface Markup Language (GIML)
- a domain independent interface object hierarchy that is based on a canonical interface object naming scheme

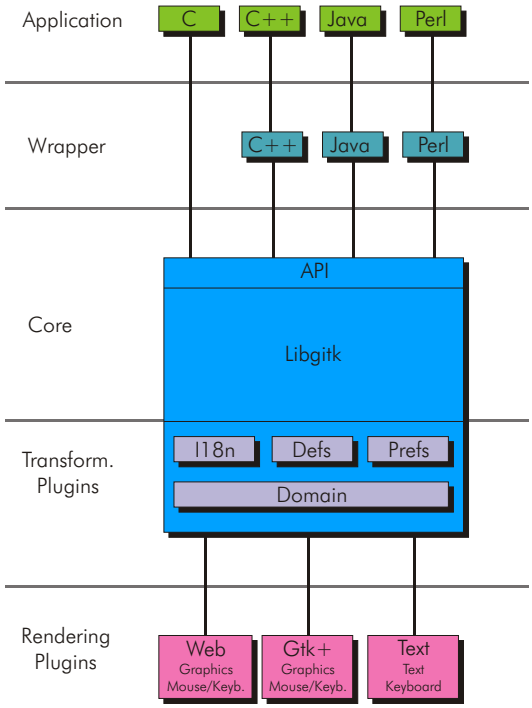


Figure 4: GITK architecture

The architecture part as shown in figure 4 has been implemented as a multi-layered software system. As a major difference to approaches like UIML, GITK not requires domain specific adaption of interfaces. The required domain specific knowledge is captured in the design of the rendering component. This sounds like a more practical approach as the application developer usually not has the knowledge and resources to cover all possible target domains. When designing a rendering component specialists can be included in the development team. The architecture presented here, can be extended, to allow applications to provide hints to the domain specific adaption on demand. This would be necessary when a generic solution is not enough.
A second key difference is that the XML interface description is used as an active dialog model. That means that the adaption processing heavily relies on XML technology such as XSLT, XPath and XML Namespaces. The advantage of this is, that there is no discontinuity in the use of technology that is processing the model in the transformation pipeline (see figure 5). The pipeline itself is maintained by the core library. This includes construction of a pipeline for a specific renderer, executing it and synchronizing both ends. At run-time the application feeds a dialog description into the pipeline. This serves as a structure onto which all variable aspects of an interface are overlaid. Then the core library initially applies all transformations to build the dialog description that the renderer understands. Each step of the pipeline adds or reconfigures aspects of the interface towards the requirements of the user. When the user works with the interface, all state changes such as navigation and data-entry are written back to the XML dialog model by the renderer (at the renderers end of the pipeline) and are synchronized with the applications end of the pipeline by the core library. This mechanism decouples the application and the interface instance even at runtime.

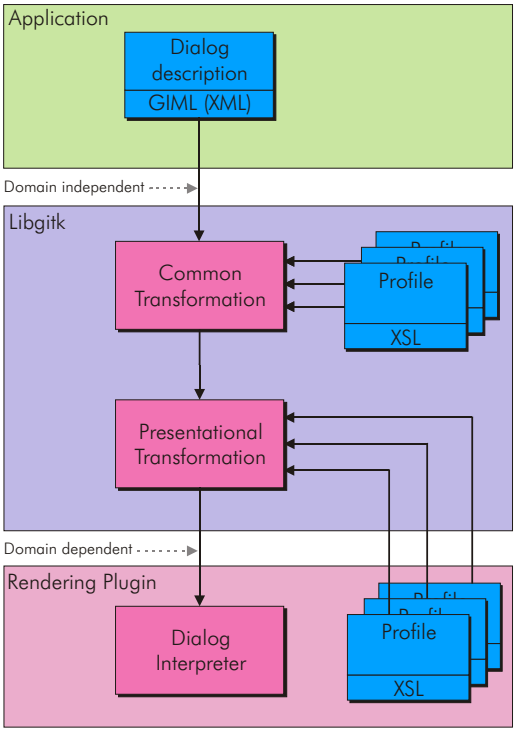


Figure 5: GITK processing pipeline

GIML is defined by a document type definition (DTD). This is currently being exchanged with W3C Schema. The markup language uses namespaces to separate the various aspects of dialogs (see section Interfaces) and namespaces do not work well with DTDs.
All interface objects are identified by a type. The type hierarchy uses only functional names. Thereby a ”push button” becomes an ”action”, as when used e.g. in the voice domain a ”push button” is not a meaningful concept. This abstraction layer allows each renderer to associate a domain dependent representation with the domain independent name.
Figure 6 shows an example dialog definition. One can see that the GIML is relative terse. It is important to note that the example only shows the input document. The application adds dynamic aspects like behavior at run-time by using the ore library API. The GITK software package comes with an introspection mechanism to look inside the XML pipeline at run-time.
This work comes with a free reference implementation. It is available as an active open-source project at http://gitk.sourceforge.net. The system is light-weight and portable. It is developed mostly in C and requires only a few libraries like glib and libxml2. It has been successfully tested on several Unix/Linux and Windows systems. The project consists of a core package, various renderers (text, gtk, opengl, phone, ...) and a set of examples. The core package comes with a

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE giml SYSTEM "http://gitk.sourceforge.net/giml.dtd">
<!-- $Id: gitkHelloUser_main.xml.in,v 1.7 2004/04/01 12:17:27 ensonic Exp $
 * @file    gitkHelloUser_main.xml
 * @author  Stefan Kost <ensonic@users.sf.net>
 * @date    Thu Jan 17 11:22:38 2002
 *
 * @brief  main dialog for gitkHelloUser.c
 * @ingroup gitkexamples
 *
-->
<giml xmlns="http://gitk.sourceforge.net/"
        xmlns:dc="http://purl.org/dc/elements/1.1/"
        xmlns:i18n="http://apache.org/cocoon/i18n/2.0">
    <dialog>
        <meta>
            <dc:title><i18n:text>query user identity</i18n:text></dc:title>
        </meta>
        <dialogwidgets>
            <dialogwidget id="Okay"/>
            <dialogwidget id="Cancel"/>
        </dialogwidgets>
        <widgetgroup>
            <label><i18n:text>identity</i18n:text></label>
            <widget id="UserName" type="characterinput_alphabetic">
                <label><i18n:text>user name</i18n:text></label>
                <disabled>true</disabled>
            </widget>
            <widget id="Sex" type="optionchoice_single_compact">
                <label><i18n:text>sex</i18n:text></label>
                <options>
                    <option><i18n:text>male</i18n:text></option>
                    <option><i18n:text>female</i18n:text></option>
                </options>
            </widget>
        </widgetgroup>
    </dialog>
</giml>
```

Figure 6: GIML dialog example

browser based management console, that allows to inspect the internals of the system and to simulate changes in the adaption requirements. [Kost, 2003]

CONCLUSION

The article started with a theoretical foundation. The terms related to interaction and adaption have been precisely defined. The problem analysis showed that the adaption problem has a multi-dimensional nature. This finding even more justified the separate exploration of *adaption infrastructure* and *adaption methods*. Then the objective of adaption - the interface - has been covered.

In the previous section a new adaption infrastructure that is able to integrate all kinds of adaption methods has been introduced. It is important to note that not only the XML language as such solves the problem of an abstract interface architecture. The interplay of language and architecture is what provides a flexible system. GITK reaches this goal by its pipeline concept and the intensive use of XML technology. The presented solution meets the requirements to design a holistic approach towards adaption. The included examples show the adaptability and the partial adaptiveness. The choice of examples outlines the kind of applications the GITK approach is useful for - administration tools, information management (CMS,PIM) software - all applications where a highly available clean interface matters more than a polished interfaces presentation. A second target group are rapid prototyping systems, as for these GITK can act as a interface prototype run-time environment. It would be interesting to research if a GITK interface description can be generated from an UMLi (Unified Modeling Language for Interactive Applications) model or even from a XML schema definition [Norman W. Paton and Paulo Pinheiro da Silva, 2002, Sperberg-McQueen and Thompson, 2004].

The current state of the project mainly affects software development and not yet the user, as it focuses on the adaption infrastructure and not the adaption methods.

FUTURE

To turn adaptable applications into adaptive systems *user profiles* are needed. User modeling and the ongoing technological development contribute towards that. Furthermore future devices will have more *sensors* to read from the environment.

Integrating such dynamic profiles and related adaption methods into the GITK infrastructure will extend the solution towards more kinds of applications.

REFERENCES

Dix, A. J., Finlay, K. E., Abowd, G. D., and Beale, R. (1997). *Human-Computer Interaction*. Prentice Hall, Pearson Education Limmited.

Dubinko, M., Klotz, L. L., Merrick, R., and Raman, T. V. (14 October 2003). Xforms 1.0. http://www.w3.org/TR/xforms/ , http://www.w3.org/MarkUp/Forms/.

Fink, J., Kobsa, A., and Nill, A. (1996). Useroriented adaptivity and adaptability in the avanti project. http://citeseer.ist.psu.edu/fink96useroriented.html.

Fink, J., Kobsa, A., and Nill, A. (1997). Adaptable and adaptive information access for all users, including the disabled and the elderly. In *Proceedings of the Sixth International Conference UM97*. Springer Verlag, Wien, New York.

Hyatt, D. (30 March 2000). The xptoolkit architecture. http://www.mozilla.org/xpfe/xptoolkit/index.html.

Kost, S. (2000-2003). Gitk - generalized interface toolkit. http://gitk.sf.net.

Norman W. Paton and Paulo Pinheiro da Silva (February 27, 2002). Umli - unified modeling language for interactive applications. http://www.cs.man.ac.uk/img/umli/index.html.

Pfaff, G. E. (1985). User interface management systems. In *EurographicSeminars*. Springer Verlag, Berlin Heidelberg New York Tokyo.

Phanouriou, C. (1999,2000a). Uiml - user interface markup language. http://uiml.org/.

Phanouriou, C. (2000b). *UIML: A Device-Independent User Interface Markup Language*. PhD thesis, Virginia Polytechnic Institute and State University.

Sperberg-McQueen, C. M. and Thompson, H. (17 Mar 2004). Xml schema. http://www.w3c.org/XML/Schema.

Stary, C. (1996). *Interaktive Systeme*. Friedr. Vieweg & Sohn Verlagsgesellschaft mbH.

various contributors (1992). A metamodel for the runtime architecture of an interactive system. *UIMS Tool Developers Workshop 1992: In SIGCHI Bulletin. 24(1). pp 32-37.*

Extensibility and Reusability of Web User Interface Components using XICL

Jair C Leite, Lirisnei Gomes de Sousa
DIMAp - Department of Informatics and Applied Mathematics
Federal University of Rio Grande do Norte
Natal, RN, 59078-970, Brazil
+55 84 215 3814
jair@dimap.ufrn.br, lirisnei@lcc.ufrn.br

ABSTRACT
Markup languages have proven successful and relatively easy at describing application user interfaces (UI). However, the most of markup user interface languages does not provide extensibility and reusability facilities to the creation of new and more powerful Web (browser-based) User Interface components. Developing new UI components using DHTML is a very hard work because of the lack of standardized models and application programming interfaces. Our work proposes XICL, an extensible XML-based markup language to user interface developments. XICL is a UI description language that provides extensibility and reusability of UI components. Using the XICL, it is possible to describe user interfaces and to develop new UI components to browser-based software applications. This language also defines a description format and a semantic model that standardizes UI components development to promote reusability and extensibility. The output of the XICL translation is a DHTML code – the W3C recommended technologies – that can run in every browser that follows that recommendation.

Keywords
User Interface Description Languages, User Interface Component, XML-based User interface.

INTRODUCTION
Web systems are characterized to have a browser-based user interface (UI). The WWW Consortium recommended technologies to user interface development – HTML, Cascading Style Sheets (CSS), Document Object Model (DOM) and ECMAScript-based languages – provides a very limited set of browser-based UI components [16,9]. To improve the interactivity of these systems it is necessary to develop new UI components that provide more advanced interaction techniques such as *pop-up menus, dialog boxes, toolbars*, *toolboxes* and others. There are some approaches to the development of UI components, but they require specifics plug-ins and/or operating platforms. In order to achieve more portability, the UI could be developed using the W3C recommended technologies. However, it is a very hard work to develop new UI components using DHTML because of the lack of an underlying component model and standardized APIs (Application Programming Interfaces). There are also no reusability and extensibility mechanisms to component composition.

There are a lot of XML-based languages to describe user interfaces with many different purposes. Most of them, such as UIML [11], XIML [12] and AUIML [4], were designed to generate a concrete user interface that runs in multiples target platforms – desktop PC, notebooks, mobiles phones, PDA, pocket PC, etc. They are concerned in how a UI component can be generated to run in many target platforms, but not in allowing programmers to develop new UI components.

XICL (*eXtensible user Interface Component Language*) is an extensible XML-based markup language to describe Web UIs and components. UI components are developed in XICL from HTML elements and others XICL elements. It was designed to provide a standardized way to UI component-based development and to promote reusability and extensibility. The extension mechanism allows the definition of new XICL components by modifying the structure, properties, methods or events of previous components to generate more powerful ones. The new component, the XICL component, can be reused in different situations, improving development productivity.

In the next section, we discuss approaches to Web UI Component development. Section 3 presents the XICL using a simple example. Section 4 discusses the XICL in the context of others XML-based User Interface Description Languages (UIDLs) and we illustrate it comparing with a UIML version of the example of section 3. The last section presents our conclusions.

APPROACHES TO DEVELOP WEB UI COMPONENTS
In the component-based software development (CBSD) paradigm, applications are composed from diverse software components (building blocks) [15]. This kind of binary component could be integrated at both design and run time

to compose a software application independently of the programming language and computing platform.

Our concept of Web UI component is not that of a binary software component used by the CBSD community. By Web UI component we mean a software object that interacts with the user via a browser. It is rendered by the browser and responds to user events dispatched by the browser. Using the DHTML technologies, a Web UI interface component could be created by a HTML tag or as a DOM object using a script language. We are concerned only with components at this level of development, regardless of their binary implementation details.

There are several approaches to the development of Web UI components. The main software industry solutions to the development of UI components are the Microsoft IE WebControls, the Macromedia Flash MX 2004 and the Sun Java Applets. The Internet Explorer WebControls are a collection of ASP.NET server controls that generate HTML content that renders in all commonly used browsers [10]. A TreeView, ToolBar, MultiPage, and TabControl user interface are included in the Internet Explorer WebControls. There are others solution from Microsoft but they are also based on the .NET technologies that are specific for Windows platform.

The Macromedia Flash MX 2004 is a proprietary and closed technology to the development of Web UI Components [2]. It provides a Component Architecture that uses the ActionScript 2 language to define Classes and Interfaces for Web components. However, the development of the components and their execution in the resulting user interface requires a proprietary plug-in that is not available to all operating platforms.

In the Java-based technologies, UI components could be developed in the Java language using some specific development framework and API (e.g. Java Swing). The running time UI components are Java Applets and they require the Java Virtual Machine to be executed in a browser. Applets are a powerful and flexible solution but they are limited to the Java development and operating platforms.

XML-based UI Component Languages

There are several markup languages to binary component composition. The Component Markup Language (CoML) is an XML application for binary composing software components [5]. The Bean Markup Language (BML) is an XML-based component configuration or wiring language customized for the JavaBean component model [1].

Our approach is different from traditional composition languages (script or markup) that act as only a glue to binary software components. XICL is a language to define non-binary UI components and also to describe the UI by structuring XICL and HTML elements.

There are also others XML-based languages used in Web development. WSUI attempts to standardize components as web services by defining a web component model that couples network services with interaction and presentation information [3]. The components can be dynamically embedded into container applications at run-time by non-developers. WSUI is programming language independent and uses standards-based technologies. It also supports multiple target display languages but the components are implemented entirely via server-side technologies.

WSUI and XICL have different purposes but can be integrated in a complementary way. WSUI defines web services in the server-side that should implement some computation. The user interaction input and output is done using standard http protocol. The output of a WSUI component could be in several target languages such as WML or HTML. In this way, it could be also a XICL output.

MXML, the Macromedia Flex Markup Language, is an XML-based markup language that is based on two popular development paradigms: markup languages and object-oriented programming languages [7]. MXML includes a richer set of UI components such as DataGrid, Tree, TabNavigator, Accordion, and Menu. MXML could be used to declaratively lay out these components. But the most significant difference is that MXML-defined user interfaces are rendered by Flash Player. To describe the behavior of the UI it is necessary to code event handlers using the ActionScript programming language. ActionScript is an object-oriented programming language that handles the user interactions with the application. It is also possible to extend the MXML tags and create new components. So, the main limitation of MXML as a markup component language is that the UI components only run in the specific Macromedia Flash technology.

THE XICL LANGUAGE

XICL is a language do describe a user interface and also to describe new UI components by reusing and extending previous ones. XICL is based on HTML elements but it could be adapted to be based on any component technology that maintains the same underlying conceptual model. Its syntax is based on XML, HTML and ECMAScript and also follows the Document Object Model. Our intention is to provide a familiar syntax to Web system developer.

The development process using XICL is described as follows and the figure 1 illustrates it. The developer specifies the UI in XICL using the text editor. He/she could reuse a XICL component from the library. The interpreter analyzes the XICL code and generates the resulting UI in DHTML code. The resulting DHTML code could run in all browsers that follow W3C recommendations.

New UI components are described in XICL using HTML elements and XICL components by reuse and extension mechanisms. A XICL component could be composed of reusable components stored in a library (XICL Lib) or it can extend an existing component modifying specific properties.

The XICL Studio is a basic environment composed of an editor, a library of components (XICL Lib) and the XICL interpreter. Using the editor it is possible to edit the description of the user interfaces or the components. When editing components the developer should store its source code in the XICL lib to be reused in future user interfaces.

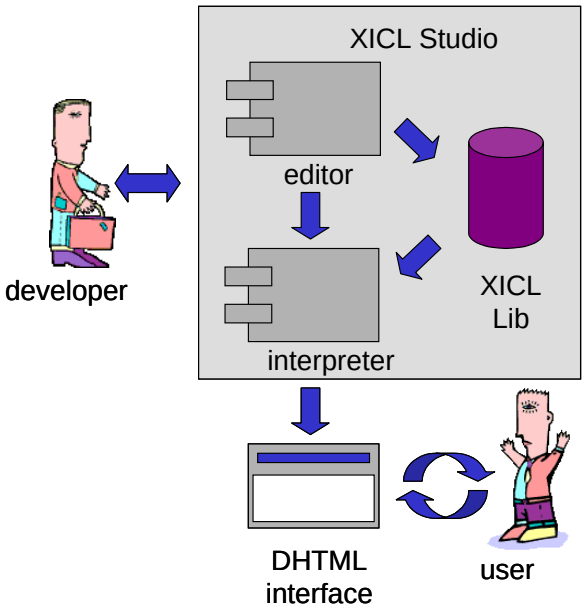


Figure 1: Developing UI using XICL

The layout of the user interface is defined as in normal DHTML user interface using CSS to associate style to XICL elements.

The reuse of XICL components can be done by importing a component from XICL Lib. The developer specifies the component by using the *import* statement. The interpreter joins the source code of the component to that of user interface and translates them into a DHTML code.

The XICL Interpreter

The XICL Interpreter is a free and open source tool that was developed to translate the XICL source code into XHTML source code.

The XICL Interpreter has three modules, namely the *XICLTranslator*, the *ComponentManager* and the *JSSourceManager*. The XICLTranslator is responsible to control the others modules and to require its services. It receives the original XICL document to be translated and return a HTML source. The components are processed by the ComponentManager module. It verifies if there are errors in the components description. The JSSourceManager works concurrently to the translation process generating the final JavaScript source code. The Javascript code controls the UI dynamics.

The translation process has three steps: *document validation*, *components importation* and *document translation*. The XICL Interpreter receives the file path and then it takes the document file and verifies if it is a valid XML document. If the document is not valid, the process stops. If the document is valid, the ComponentManager module imports the remote components. If the main document imports the lib1 and lib1 imports the lib2, the ComponentManager will import every lib1 components and the every lib2 components. If the lib2 has a component with the same name of any component in the lib1, the component from the lib2 will not be imported. For instance, if the lib1 has a component called WINDOW and the lib2 also have a component with the same name, only the former WINDOW component will be imported from the lib1.

After the document validation the XICL Interpreter can start the translation. This step consists in translating recursively all elements that are on the interface description document.

A Simple Example

To illustrate the characteristics of XICL we show a simple example of a UI description that reuses XICL components. The example is very simple and could be created only with DHTML. However, the example shows how the reuse and extension mechanisms simplify the development process. They provide a simple description. We choose this example also to compare it with other XML-based UIDL.

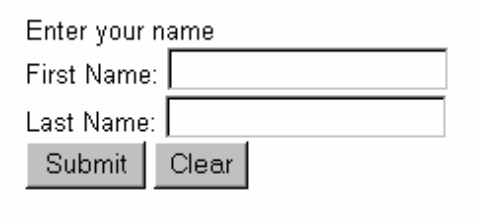


Figure 2: The UI view

The figure 2 shows the UI rendered in a browser. The interface has two text boxes that allows the user to enter information (**First Name** and **Last Name**) and two buttons (**Submit** and **Clear**). When the user click on the Submit button a dialog box (**Confirm**) is displayed asking if the user wants to open a new window.

The XICL description

A XICL description can contain zero or more component descriptions and one or zero interface descriptions. It also can make reference to components that were defined in another XICL documents. In our example, the XICL code is described in two files, one for the UI description and other for the component description. However, it is possible to describe both the user interface and its components in a single file.

```
1.  <XICL>
2.    <IMPORT href="lib1.xml">
3.
4.    <INTERFACE>
5.      <p/>
6.      <span>Enter your name</span>
7.      <p/>
8.      <TEXT id="t1" length="8" label="First Name: " />
9.      <p/>
10.     <p/>
11.     <TEXT id="t2"length="8" label="Last Name: " />
12.     <p/>
13.     <SUBMIT VALUE="Submit"
      onclick="mess1.show()"/>
14.     <RESET VALUE="Clear"/>
15.
16.   <ConfirmBox id="mess1" title="New Window"
      top="200" left="200"
      onConfirm="window.open('http://uiml.org',
      'newWin');" >
17.       This link opens another window. Would you like
      to proceed?
18.   </ConfirmBox>
19.
20.   </INTERFACE>
21. </XICL>
```

Figure 3: The XICL description of the UI

The UI description is shown in figure 3. The UI is developed making use of a XICL component defined in the lib1.xml file and imported with the IMPORT element (line 2, figure 3). The INTERFACE element is used to describe the user interface structure and behavior. The HTML components and its associated events are specified using the XHTML syntax (a XICL document is a XML document). Our example has four static components that are instances of TEXT, SUBMIT and RESET. It has also a dynamic component, the **ConfirmBox**. The **ConfirmBox** is invisible in the beginning of the execution and is just displayed when triggered by the **onConfirm** event that call the **window.open()** function. The **onConfirm** event occurs when the user clicks on the SUBMIT component. The **ConfirmBox** is shown in figure 4. If the user clicks on the **Cancel** button the component will be closed.

The definition of the components **TEXT**, **SUBMIT**, **RESET** and **ConfirmBox**. is shown in figure 5. A component description has four parts: *properties*, *structure*, *events and methods*. The description begins with the <COMPONENT name=…> tag informing the component name and ends with the </COMPONENT> tag. The *name* attribute is used to reserve a *namespace* to the component so the component can be used on the interface description only putting its name between tag marks (e.g., <TEXT>).

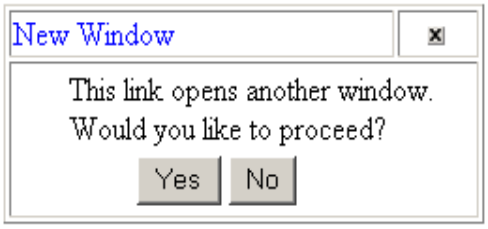


Figure 4: The ConfirmBox

The **TEXT** component is defined as a composition of a simple text (the label) with a HTML input element. The **STRUCTURE** element (line 4, figure 5) structures them with a **span** (line 5, figure 5) element that joins the variable label (**$label**) with the **input** element. The value of label is assigned when the instance of the component is used in the interface definition (lines 8 and 11, figure 3). In our example we do not use the **property** element just to keep it simple.

The **EVENT** element of the SUBMIT component (line 13, figure 5) defines the **onclick** event that is triggered by the submit button element and calls a function that should be defined when the component is used. The figure 3, line 13, shows an instance of the SUBMIT component that calls the **mess1.show()** when the user clicks on it.

Components can be defined extending others components. In order to do it we use the *extends* attribute. The **ConfirmBox** component is a window that shows a confirmation message to the user. This component is similar to the *JavaScript Confirm* and was developed just to demonstrate the language characteristics. The **ConfirmBox** extends the Window component, which must be described in the library.

The **structure** element can contain any HTML or XICL components. In the **ConfirmBox** component, there are TABLE, FORM and INPUT elements. It is also possible to describe a generic component that can be replaced by any other one (<COMPONENT ref="ANY"/>, figure 5, line 26), which should be defined when it is instantiated. The instance of the **ConfirmBox** component in figure 3 defines the string "This link…" to replace the generic component (figure 3, line 17). This is a powerful characteristic of XICL allowing the structure of the component to be defined just when used.

```
1.  <XICL>
2.   <IMPORT href=”lib2.xml”>
3.    <COMPONENT name="TEXT">
4.      <STRUCTURE >
5.       <span>$label</span><input type="submit"
    maxlength="$length" >
6.      </STRUCTURE>
7.    </COMPONENT>
8.    <COMPONENT name="SUBMIT">
9.      <STRUCTURE >
10.      <span>$label</span><input type="submit"
    maxlength="$length" name=”subBnt”>
11.     </STRUCTURE>
12.     <EVENTS>
13.      <EVENT name=”onclick”
    trigger=”subBtn.onclick” function=”$onclick” >
14.     <EVENTS>
15.   </COMPONENT>
16.   <COMPONENT name="RESET">
17.     <STRUCTURE >
18.      <span>$label</span><input type="submit"
    maxlength="$length">
19.     </STRUCTURE>
20.   </COMPONENT>
21. <COMPONENT name="ConfirmBox"
    extends="Window">
22.    <STRUCTURE>
23.     <table width="100%" border="0" >
24.       <form>
25.         <tr>
26.           <COMPONENT ref="ANY"/>
27.         </tr>
28.         <tr>
29.           <td align="right">
30.             <input type="button" name="bOk"
    value=" Yes " />
31.           </td>
32.           <td align="left" >
33.             <input type="button" name="bCanc"
    value=" No " onclick=" $id.close();" />
34.           </td>
35.         </tr>
36.         </form>
37.      </table>
38.     </STRUCTURE>
39.
40.   <EVENTS>
41.      <EVENT  name="onConfirm"
    trigger="bOk.onclick;” function="$onConfirm;
    $id.close();"/>
42.    <EVENT  name="onCancel"
    trigger="bCanc.onclick;"
    function="$onCancel;$id.close();"/>
43.   </EVENTS>
44. </COMPONENT>
45.  <XICL>
```

Figure 5: The description of XICL Components

COMPARING XICL WITH OTHER XML-BASED UIDLS

XICL has different purposes than those of others XML-based UIDLs, such as UIML [11], XIML [12], AUIML [4], XUL [6] and Xforms [8]. XICL is an implementation level language. Unlike XIML and AUIML, XICL does not concern with providing an abstract description of the user interface. Like XICL, XUL is an implementation level language but it requires some technologies introduced by Mozilla. Xforms is also an implementation level language, but its goal is different from that of XICL. It provides advanced capabilities to HTML form elements. For a more detailed discussion see [14]. In the next question we compare XICL and UIML for an illustrative purpose.

None of these languages provides the extensibility and reusability of browser-based UI components. By browser-based we mean DHTML-based. Because of this characteristic, it is possible to develop all kind of user interfaces that could be done with DHTML. However, XICL provides the benefits of component-based software such as better productivity and maintainability.

We now review XICL using the criteria proposed by Souchon and Vanderdockt [13]. In that work, the authors propose two groups of criteria to compare the characteristics of XML-compliant UIDLs. We now discuss XICL properties according to those criteria as a way to compare it with the others languages.

In the first group, the language is compared by the following criteria:

- **Component models**: The aspects of the UI that can be specified in the description - task model, domain model, presentation model and dialog model. XICL provides constructs to describe the static aspects of the presentation using the structure and properties elements and the dialog aspects using a combination of event, method and script elements.
- **Methodology**: Different approaches to specify and model user interface. XICL is an implementation level language. It helps developer to implement the UI faster and easier than with only DHTML.
- **Tools support**: The languages and tools that support specification, translation and rendering. A XICL description can be produced using a plain text editor or an XML editor. An interpreter implemented in Java translates from XICL to the final DHTML code.
- **Supported languages**: The programming language to which the language can be translated. The current version of the XICL only can be translated to DHTML.
- **Platforms**: The computing platform on which the language can be interpreted and rendered. The XICL interpreter is implemented in Java and it generates a DHTML code. The UI generated from a XICL description can be rendered in any standard browser.

- **Target**: The context of use - the user model, the environment model and the platform model. XICL is multi-platform. The current version of XICL was originally designed to the context of browser-based UI.

The second group of criteria's is the following:

- **Abstraction level**: The different levels to describe the UI. XICL describes the user interface on the instance level.
- **Amount of tags**: XICL has just 13 core tags to describe the interface and components. New tags corresponding to new UI elements could be developed since that one of the main goals of the language is the extensibility of components.
- **Expressivity of the language**: Capability to express concepts and the usability to manipulate them. We have no formal evaluation of those aspects in XICL. However, since XICL has extensibility mechanisms, it has a high expressivity. It also follows the HTML syntax so we believe it is easy to learn and use.
- **Openness of the language**: The possibility to extend concepts or tags. As we said before, extensibility mechanism is one of the important aspects of XICL.
- **Coverage of concepts**: The concepts a language introduces. XICL work with two main concrete-level concepts: interface and component. A component is defined using the concepts of **structure, property, event** and **methods**. No abstract-level concept is covered by XICL.

A UIML DESCRIPTION OF THE EXAMPLE

The example presented in the last section was extracted from the UIML[1] (*User Interface Markup Language*) web site. Our intention was to compare how a simple example written in an XML-based UIDL could be written in XICL in order to analyze some basic differences. We choose UIML because it is one of the first XML-based UIDL and there are many available tools and examples.

UIML has a different purpose than XICL. It was designed to allow platform independent UI development – hardware devices, operating systems and programming languages [11]. From a unique specification in UIML it is possible to generate User Interfaces to several platforms in different programming languages. The main purpose of XICL is to provide extensibility and reusability mechanisms. However, in our example we can see the UIML and XICL descriptions generating the same final user interface. So we consider that they were used with a similar purpose.

Figure 6 shows the UIML description of the UI in figure 2 to be rendered by a browser. The UIML code is divided into three parts, which are represented by three elements. The **structure** element describes the UI visual structure. It is composed by HTML and UIML elements such as SPAN, TEXT, SUBMIT and RESET. The UIML elements are pre-defined and cannot be created by the developer. This is one of the main differences of XICL to UIML and other UIDLs.

```
1.  <structure>
2.    <Html>
3.      <Body>
4.        <Form>
5.        <P/>
6.        <Span content="Enter your name"/>
7.        <P/>
8.        <Span content="First Name:"/>
9.        <Text maxlength="8" />
10.       <P/>
11.       <P/>
12.       <Span content="Last Name:"/>
13.       <Text maxlength="8"/>
14.       <P/>
15.       <Submit id="SubmitButton"
    value="Submit"/>
16.       <Reset value="Clear"/>
17.       </Form>
18.     </Body>
19.   </Html>
20. </structure>
21.
22. <behavior>
23.  <rule>
24.   <condition>
25.     <event class="OnClick" part-
    name="SubmitButton"/>
      </condition>
26.
      <action>
27.     <call name="form.submit"/>
28.   </action>
29.  </rule>
30. </behavior>
31. <peers>
32.    <presentation ... />
33.    <logic>
34.      <d-component id="form"... >
35.        <d-method  id="submit" ...  >
36.         <script type="text/Javascript">
37.                function checkBeforeProceeding () {
    if
    (confirm("This link opens another
    window.          Would you like to proceed?"))
    window.open('http://uiml.org', 'newWin'); }
38.         </script>
           </d-method>
39.      </d-component>
40.   </logic>
41. </peers>
```

Figure 6: UIML code

[1] The example is available at the UIML Web site http://www.harmonia.com/products/html/examples/scripts.htm.

The UI dynamics is described by the **behavior** element. It consists of rules that associate condition to actions. The **rule** describes an **action** that is called when the **event** occurs. The **peers** element has the **logic** element that is used to describe the **script** function and to associate it with a submit method (**d-method**) of the form element (**d-component**).

Considering that in this example the languages were used to generate the same user interface, we analyzed the main differences between them.

- UIML is a good device independent language. It is possible to generate a concrete user interface to several targets using the same description. XICL does not have these characteristics.
- UIML has a limited set of pre-defined components to describe a user interface. So, the developer should reuse just these pre-defined components. In XICL, the components are described and stored in a library so it can be reused later. Applying the reuse mechanism, the UI description source code in XICL (figure 3) is shorter than in UIML (figure 6).
- XICL also allows the definition of new components that can contain (is composed of) a generic component that is defined just when reused to describe a user interface. UIML has no such mechanism.
- The description of the UI behavior is described in a similar way. They are based in the definition of events and methods. The methods are script functions that are triggered by events. The main difference is that in XICL is possible to define new events when defining new UI components.
- The UI presentation details in UIML are described by the presentation element whereas in XICL it is described using CSS.

CONCLUSION

Developers need always to construct new UI components to achieve more system usability. The W3C recommends the use of DHTML technologies in the client-side user interface to increase application portability. However, DHTML only provides a few basic UI components such as *button, drop-down menu, text fields, check-box, radio-button,* etc. and developing them in DHTML is a very hard work. Also, there are no models and standards to the development of DHTML UI components.

XICL is a language to User Interface development by specifying its structure and behavior in an abstract level than using only DHTML. The main objective of XICL is to provide mechanisms to the creation of new Web UI components by extending and reusing others components.

XICL is based on the XML syntax and it follows a basic component model to provide a well-structure code. The XICL code smoothly integrates with DHTML technologies promoting interoperability.

UI development in XICL can be done using the XICL Studio environment. This basic environment provides a simple editor, a library of components and an interpreter that translate XICL code into DHTML code. The final user interface is implemented using DHTML technologies and can run in common Web browsers.

XICL does not address the purpose of device independence as many others UIDLs. However, it is possible to translate XICL to others implementation languages providing it with multi-target capability. It is necessary to develop an interpreter to translate XICL to a specific target language. For instance, we could develop a WML interpreter to allow XICL to be used in the description of mobile user interfaces or a Java Swing interpreter to generate a GUI.

It is also possible to integrate XICL with others UIDLs since they are specified using XML. We could define the elements of an UIDL in XICL. We could also provide the XICL mechanism to define new components in others UIDLs.

XICL is based on HTML elements but it could be adapted to be based on any component technology maintaining the same underlying conceptual model.

ACKNOWLEDGMENTS

We thank PRH22-ANP/MCT for the partial financial support to this work.

REFERENCES

1. AlphaWorks, Bean Markup Language, Update: November 24, 1999, http://www.alphaworks.ibm.com /formula/bml
2. Anbar, Waleed Exploring Version 2 of the Macromedia Flash MX 2004 Component Architecture, http://www.macromedia.com/devnet/mx/flash/articles/, accessed in September 20, 2003.
3. Anuff, E., Chaston, M., Moses, D & Kropp, A. Web Service User Interface (WSUI) 1.0 Working Draft – 31 October 2001. http://www.wsui.org/doc/20011031/WD-wsui-20011031.html
4. Azevedo, Pedro; Merrick, Roland & Roberts Dave, OVID to AUIML - User-Oriented Interface Modelling TUPIS'2000, Towards a UML Profile for Interactive Systems Development. York, UK, 2000.
5. Birngruber: A Software Composition Language and Its Implementation in: Bjorner Dines, Broy Manfred, Zamulin Alexandre V. (Eds.): Perspectives of System Informatics (PSI 2001), July 2001, LNCS 2244, Springer, 2001, pp. 519-529.
6. Boswell, David; King, Brian; Oeschger, Ian; Collins, Pete and Murphy Eric. Introduction to XUL. In Creating Applications with Mozilla. O'Reilly, September 2002.
7. Coenraets, C. An Overview of MXML, the Macromedia Flex Markup Language. Available at http://www.macromedia.com/devnet/flex/articles/paradigm.html, accessed in February 21 2004.

8. Dubinko, Micah; Klotz Jr., Leigh, Merrick, Roland; and Raman, T. V. XForms 1.0 W3C Working Draft 21-August-2002. in http://www.w3.org/TR/xforms/ accessed in March 22, 2003.

9. Goodman, D. Dynamic HTML – The Definitive Reference. O'Reilly, 1998.

10. Microsoft Corporation. "Internet Explorer WebControls Reference". The MSDN Library, in http://msdn.microsoft.com/library/, accessed in March 22, 2003.

11. Phanouriou, Constantinos "UIML: A Device-Independent User Interface Markup Language." Phd Thesis, Virginia Polytechnic Institute, Blackburg, Virginia, 2002.

12. Puerta, Angel and Eisenstein, Jacob "XIML: A Universal Language for User Interfaces", Reale Software, 2001. in http://www.ximl.org/, accessed in March 22, 2003.

13. Souchon, N., Vanderdonckt, J. *A Review of XML-Compliant User Interface Description Languages.* Preliminary Proc. of 10th Int. Conf. on Design, Specification, and Verification of Interactive Systems DSV-IS2003 (Madeira, 4-6 June 2003), Jorge, J., Nunes, N.J., Falcão e Cunha, J. (Eds.), Lecture Notes in Computer Science, Vol. 2844, Springer-Verlag, Berlin, 2003.

14. Sousa, L.G, Leite, J.C XICL - An Extensible Markup Language for Developing User Interface and Components. *Proceedings of the fourth International Conference of Computer-Aided Design of User Interface CADUI'04*. Island of Madeira, Portugal, 2004.

15. Szyperski Clemens: Component Software - Beyond Object-Oriented Programming. Addison-Wesley. 1997.

16. W3C, in http://www.w3c.org, accessed in February 11, 2004.

Abstract User Interface Markup Language

Roland A. Merrick
IBM Ease of Use
P.O. Box 31, Birmingham Road
Warwick, CV34 5JL, UK
roland@uk.ibm.com

Brian Wood
IBM Server Group
Rochester MN, USA
bowood@us.ibm.com

William Krebs
IBM Software Group
RTP, NC, USA
krebsw@us.ibm.com

ABSTRACT

The ever increasing variety of devices available to users means that it is not economically viable to develop tailored user interfaces for each device. This paper describes an XML Vocabulary, Abstract User Interface Markup Language (AUIML), which has been developed to allow some classes of interactive application to be developed once and adapted to run on a wide variety of device types. The language does not take the lowest common denominator approach while using abstraction to describe the user interface. This allows device dependent adaptation to take place when rendering.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features – abstract data types, constraints, control structures.

H.5.2 [**Information Systems**]: User Interfaces – Graphical User Interfaces, Interaction styles (e.g., commands, menus, forms, direct manipulation).

Keywords

User interface, device independent, design, languages

INTRODUCTION

This paper describes an XML vocabulary, Abstract User Interface Markup Language (AUIML), which has been designed to allow the intent of interaction with a user to be defined. This is in contrast to the conventional approach that has focused on the appearance of the interaction in terms of Graphical User Interface (GUI) widgets. This "intent based" approach allows task designers to concentrate on the semantics of the interactions without having to concern themselves with which particular device types that need to be supported.

AUIML allows for device independent encoding of information needing to be exchanged between users and systems. This information, and its structure including validation rules, relationships, etc., can be encoded once, independent of the target devices, and subsequently rendered on different form factors using device dependent rendering. The latter is accomplished with an AUIML renderer on each device.

AUIML is a Model-View-Controller (MVC) language. An AUIML renderer provides a view and controller model, paired with the data model defined in the application's XML, to protect the application from any differences between devices.

AUIML was also designed to satisfy related requirements for internationalization and accessibility. The support for these features are not described in detail in this paper but were simple to accommodate within the intent based approach taken.

This work originated from an IBM® internal project called Druid [1] and was designed to simplify the development of applications to perform system and user administration, and hence supports quite sophisticated tasks. All of the relatively simple "form type" applications are encompassed by this work, such as those defined using HTML forms.

PROBLEM

The advent of Pervasive Computing has introduced a broad spectrum of devices and form factors on which applications can potentially run. While there has been some convergence in recent years in technologies that can be used to develop user interfaces for these devices, there are still some significant differences. The platform specific toolkits vary considerably. Web browsing technologies have, to an extent, started to converge on an HTML/XHTML variant, but they still have significant differences and authors still tend to write separately for each class of device.

Many users today use several different devices, such as mobile phones, laptops, and hand held devices, each with their own user interface and applications. It is the user who must shoulder the burden of learning each, and maintaining data that exists in different forms and formats on each device. Obviously this is not ideal from the user standpoint, since learning requirements increase from inconsistencies

and maintenance increases from data redundancies. These, in turn, slow the rate of adoption of new technologies that could benefit the user, and narrow the potential market for new devices to early adopters and technologists.

From the development perspective, different challenges exist. In the 1990's cross platform applications were quite narrowly defined as running on at least two different operating systems. This simple definition has applied to both client and server based software.

Even with similar operating systems and similar hardware platforms, it has proved difficult to develop software that runs on more than one platform. One particular area of difficulty is the support for the multitude of GUIs and the APIs used to create them.

For certain classes of application this process has been greatly simplified in recent years, first by the World Wide Web and HTML, and subsequently by Java™ and its Write Once Run Anywhere (WORA) promise. Unfortunately, Write Once Run Anywhere did lot live up to its promise for user interfaces. Each form factor has different input and output technologies and devices. If screens exist, they come in many different sizes. This has led to different user interface requirements, and the necessity to develop unique code for at least the user interface on each specific form factor. This is not ideal from the developer standpoint, leads to inconsistencies and increased development costs and maintenance, and slows the delivery of new technologies to the user.

From the viewpoint of technology companies, slower adoption and increased costs increase the risk of new developments and lower profit expectations. Because of this, some promising technologies may never move from research departments to development and manufacturing. User companies do not gain the benefits of the new technologies, postponing benefits which might otherwise be realized sooner.

How does AUIML help?

AUIML is an XML vocabulary that has been designed to allow the intent of user interaction to be defined. This is in contrast to the conventional approach to user interface design which has focused on appearance in terms of GUI widgets. This intent based approach allows user interface designers to concentrate on the semantics of the interactions without having to concern themselves with which particular device types need to be supported.

The intent based approach provides a great deal of flexibility by allowing decisions on the most appropriate presentation to be taken by the device dependent renderer. For example, should a choice need to be made by a user, the intent model will merely state the list of alternatives, a default, and how many choices are allowed. This will allow a device dependent renderer to exploit the resources available on the device in the most effective manner.

Using AUIML, the same application can now be rendered on Java Swing, web servlet, web portal, PDA, and WAP devices. An example of a WAP-based cell phone rendering is shown below.

History

A foundational requirement for AUIML was to support the functionality already available in IBM's system management user interfaces. These interfaces were originally written using the C++ language and its class libraries on Microsoft Windows®.

While the widgets and functionality were rich, the application code was not portable to other platforms. The user interface leveraged features only available with the platform widgets, so its user interface was not portable.

XML was also starting to come to fruition at this time, and the IBM Rochester lab developed a new XML vocabulary, named Panel Definition Markup Language (PDML), to help build portable Java Swing applications quickly. Java allowed an application's code to easily move between platforms while XML provided a language independent way to persist the user interface definition.

XML provided a further benefit since it was easily tooled. A user interface builder was created to automatically generate the XML for a constructed user interface. PDML developers could visually develop rich user interfaces without knowledge of Java Swing or the other underlying technologies being used. While this greatly increased programmer productivity and code portability, PDML's design made it difficult to migrate to the web or pervasive devices.

PDML used Java Swing controls and properties, which were rich with features not available on other platforms. The controls were placed at exact pixel locations on the panel, which was inflexible for providing a meaningful panel to a smaller screen on a pervasive device.

The data model and controller in PDML also had multiple flaws preventing it from working on different devices. The data model operated on Java Swing objects, which were not available in pervasive or web environments. PDML required model updates be displayed on the user interface immediately, which is not always possible in pervasive and web environments. Furthermore, it required user interface events to be known to the application as soon as they happened, which is difficult in a client-server application. These requirements greatly restricted PDML's flexibility in moving to other devices.

Nevertheless, there existed hundreds of PDML panels that needed to be used on web and pervasive devices with as little impact to application code as possible. The panels needed to lose little functionality when displaying on their original device, while still remaining flexible enough to display on various devices.

AUIML

AUIML is also an MVC XML language, but its view and controller is delegated to different device dependent renderers. Each renderer handles the presentation and user interaction appropriate for its targeted device. The model is common between all device dependent renderers and protects the application code from the differences in the devices. Because application code is written in Java and AUIML, it is both platform and device independent.

Data Model

AUIML is focused more on the application's intent rather than its exact appearance. The XML vocabulary consists of many simple data types and data structures to gather and output data for the panel. AUIML does not require presentation information to be specified since the device specific renderers gather and output the data appropriately for the device. These data types are basic types that can be supported across all devices: string, number, date, time, image, audio, and boolean. The presentation for each may vary for the device, but the application using the data model does need to be aware of this.

For instance, a restricted number input field in Java Swing may display as a slider control where the user moves a marker between the various possible values. The same input field on the web or on a pervasive device may request the input using a text entry field with a description of the valid number range. In both cases, the renderer ensures the input validates correctly before persisting it to the data model. This prevents the application code from caring about how the data is collected and also frees it from ensuring the data is valid.

Data state and validation is controlled by properties on the data type specifying whether the data is visible, read-only, disabled, requires a value, or is constrained to a minimum or maximum value or length.

AUIML also has data structures for trees, tables, choices, and panel aggregations. Each of these is a collection of the basic data types. A tree is a hierarchical collection of the basic data types. A table is a tabular display of the basic data types, and choices allow a selection of one or more data type values. While a panel is a collection of basic types, a panel aggregation is simply a collection of panels.

This focus on the data allows the intent of the application to be provided without presentation information. Since the intent lacks presentation details, it is portable between the different form factors and devices.

All data objects bind to a Java Bean, which allows the application to exchange data with the common data model. The bound bean is notified at the appropriate time to load and save its data and acts as both a data source and a data store to the common data model.

The bean has the final say on whether the data is valid or not. It is possible for user entered data to comply with all validation guidelines known to the renderer, but the input will not be valid when considered in the context of other data. When this happens, the bound bean can reject the data and the renderer will request the user to modify the entry so it conforms.

Interaction

User interaction with the device is also abstracted. Events may be raised when an input field is completed, a panel is initially displayed, or the user requests an action through a button or menu selection. In each of these cases, the application code is notified of the event and can take appropriate action.

Different device renderers may defer the activation of the event until it is convenient. A web device may not be able to immediately notify the application that data entry was completed for a field. Instead, it must defer the notification until another server request is made by the client.

Actions are events with a special meaning in AUIML. They may signify that data entry is complete for the panel and data should be persisted back to the data store. They may also request help text, or signify that special action handling code should be invoked. Regardless of their task, actions are always invoked immediately within the application and never deferred.

AUIML also supports automating events based on the state of the data model. These events automatically take an action upon data elements in the model when the user changes data. The most common case is to disable or hide data elements when another data element is selected or unselected. The device can usually handle these special events without ever executing application or server side code. This allows the user interface to demonstrate automatic behaviors based on the data model.

Presentation

Developers desire to create the richest user experience possible for their targeted platforms. Each rendering device has display capabilities and guidelines that cannot or should not be matched by other rendering devices. AUIML's abstraction of the device rendering prevents the developer from controlling the device's presentation in a way that may increase the usability of the application on that platform.

For this reason, AUIML provides the application developer some presentation flexibility by surfacing predefined presentation attributes. Implementation of any particular presentation attribute by a renderer is not guaranteed, but it does allow the developer to signify their preferences for data element presentation.

The properties are only preferences, as not all devices may be capable of accommodating them. For instance, the developer may request that a date data element be presented as a calendar widget. This particular widget may not be available on a pervasive device, so a simple entry field would be displayed instead. Since data input validation and event generation behave the same in both cases, the application code does not need to know if the presentation preference was followed.

The presentation attributes are specified in a format similar to that already used for Cascading Style Sheets (CSS) and are provided independently of the AUIML XML definition. This independence allows different collections of attributes to be used for the same XML definition and keeps presentation specific attributes out of the data model. When the application is translated to other languages, the presentation attributes may be customized for each language without negatively impacting the application or its model.

Tools

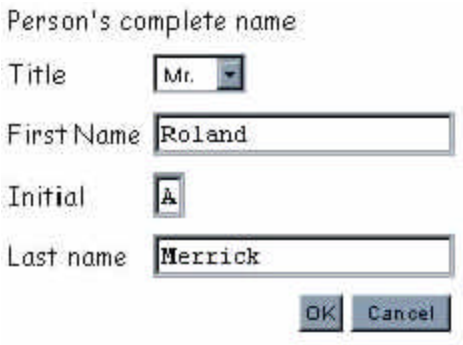
AUIML can be easily generated with a visual XML builder, much like PDML. The tooling will automatically create beans bound to data objects and generate template code for manipulating the data model. The user interface is built using an embedded version of the Swing device renderer, although other renderer types could be plugged in as well.

Properties in the tool are divided into two categories, data and presentation. Presentation properties are discouraged although they are made available if the developer has a strong display preference.

The ability to tool the user interface XML rapidly decreases the cost, time to develop, and maintenance of the user interface. The AUIML tooling allows the user to preview their work in each device renderer as they define the interface. This provides affirmation that the intent of the application is adequately migrating to the targeted device.

EXAMPLE

The following is a simple example of a panel and its rendering in HTML



The above figure is generated from the following AUIML XML definition. The XML defines a DATA-GROUP that defines the container. Each control is represented by a base data type, which has basic properties and a CAPTION. The CAPTION element provides descriptive text for data elements.

```
<AUIML VERSION="AUIML:1.2">
 <DATA-GROUP NAME="Name">
  <CAPTION>
   <META-TEXT>Person's complete name</META-
TEXT>
  </CAPTION>
  <STRING NAME="Title" ENUMERATION="OPEN">
   <CAPTION>
    <META-TEXT>Title</META-TEXT>
   </CAPTION>
   <VALID-VALUE NAME="Mr">
      <VALUE>Mr.</VALUE>
   </VALID-VALUE>
   <VALID-VALUE NAME="Mrs">
      <VALUE>Mrs.</VALUE>
   </VALID-VALUE>
   <VALID-VALUE NAME="Ms">
      <VALUE>Ms.</VALUE>
   </VALID-VALUE>
   <VALUE>Mr.</VALUE>
  </STRING>
  <STRING NAME="FirstName">
    <CAPTION>
      <META-TEXT>First Name</META-TEXT>
    </CAPTION>
    <VALUE>Roland</VALUE>
  </STRING>
  <STRING NAME="Initial" MAX-LENGTH="1">
    <CAPTION>
      <META-TEXT>Initial</META-TEXT>
    </CAPTION>
    <VALUE>A</VALUE>
  </STRING>
  <STRING NAME="LastName">
```

```
    <CAPTION>
      <META-TEXT>Last Name</META-TEXT>
    </CAPTION>
    <VALUE>Merrick</VALUE>
  </STRING>
  <ACTION NAME="OK" TYPE="COMPLETE">
    <CAPTION>
      <META-TEXT>OK</META-TEXT>
    </CAPTION>
  </ACTION>
  <ACTION NAME="Cancel" TYPE="CANCEL">
    <CAPTION>
      <META-TEXT>Cancel</META-TEXT>
    </CAPTION>
  </ACTION>
   </DATA-GROUP>
</AUIML>
```

AUIML Toolkit

AUIML has recently been released for trial usage on IBM's alphaWorks site [2]. The AUIML Toolkit includes the previously mentioned AUIML Swing renderer and AUIML HTML renderers. The AUIML HTML renderer can be used in IBM Websphere® Application Server version 5 and above.

The toolkit also contains a version of the AUIML HTML renderer that will render AUIML in IBM Websphere® Portal Server version 5. This version of the AUIML HTML renderer has implemented portal interfaces to allow the AUIML application to be hosted as a portlet. In every other respect, it is the AUIML HTML renderer.

An Eclipse-based XML builder is also included in the AUIML Toolkit. It provides What You See Is What You Get (WYSIWYG) editing of the AUIML panels using the AUIML Swing renderer. The AUIML panel must be previewed to see its presentation in the AUIML HTML renderer.

EXPERIENCES

Lessons were learned in many areas during the development of AUIML and its renderers. While the user interface definition is protected from differences between the devices, the user interface implementation by the device dependent renderers is not. Also, there are continual tradeoffs between leveraging the capabilities of a single device and keeping the programming model common between all devices.

Common Denominators

Creating an XML vocabulary that applies equally well to all devices usually requires a common denominator approach. While this approach allows the vocabulary to move laterally across the devices, it does not allow the application to leverage the deep presentation capabilities of the device. In order to maintain the presentation capabilities that PDML provided, we required a property mechanism, similar to CSS, which could be ignored by devices that did not support any single presentation property.

Conversely, using the common denominator approach without presentation customization can be a benefit. Less customization of control appearance leads to more consistent controls and behaviors across the interface and across the platforms. For this reason, the AUIML tooling and language does not require, or even encourage, the use of presentation properties. The correct level of customization is left to the user interface designer.

Renderer Consistency

The data model may be common between all devices, but the device dependent renderers must fill and interact with the data model in exactly the same way or the application will fail. Even with behavior specifications and cross-renderer communication, the various AUIML device renderers sometimes behaved differently for data validation, user messages, and program flow. While these were program defects in the renderers, the problem deserves special care and consideration for all interpreted user interface XML languages.

Interaction levels on different devices differ greatly. A web or pervasive device may not always be connected to the application and may only communicate during request-response cycles. This was the case for the AUIML HTML device renderer and caused it to diverge from Swing in some interaction behaviors.

User events in Swing could be applied to the model immediately and the changes in the model would be surfaced back to the user interface immediately. HTML requires a queuing or delaying of interactions until the next request-response cycle. If a model change was made on the server, the client would not become aware of this change until the next server request. Since the clients are disconnected from the server, there is no guarantee there will even be another request.

The AUIML HTML renderer queued events that did not need to be issued immediately and queued all updates to the user interface. The user could force these events to happen immediately by pressing a button on the HTML user interface. AUIML takes into account the interactivity of different devices and will provide additional widgets such as this button when necessary. This may add artifacts to the user interface, but it allows a single application to run across environments with less regard for interaction differences.

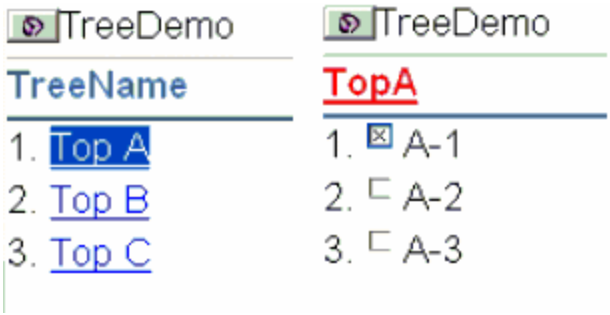
Pervasive Devices and Small Form Factors

Tables and trees were the most difficult controls to render in a constrained space for PDA and WAP devices. The device dependent renderer chose to break down the presentation of these data elements into index, row, and record views. For example, the user can navigate through grouped table rows using the numeric keys on a cell phone.

The rows are grouped into a set of row indices for navigation as shown below.

When selecting a group of rows, the row index lists the key text for that row. After choosing the desired row, the details for that row are shown. Buttons are added by the device dependent renderer to allow the user to move forward and backwards in the navigation hierarchy.

Trees show nodes that contain child node elements. When a node is selected, the child nodes are displayed. A restriction for this renderer is that true model selection of nodes is not allowed since the selection operation actually maps to an expansion function.



XForms

The experience gained during the design of the language and the development and deployment of the various AUIML runtime environments provided valuable experience for IBM to draw on in its participation in the World Wide Web Consortium XForms [3] and Device Independence [4] Working Groups.

A full comparison between AUIML and XForms [5] is not appropriate in this paper but some similarities are apparent. The use of abstract user interface widgets and offering the user the opportunity to choose from a set of alternatives, <CHOICE> in AUIML and <xforms:select> in XForms, is one similarity. Abstract event handlers, <WHEN-SELECTED> in AUIML and <xforms:action ev:event="xforms-select"> in XForms, is also similar between the two technologies.

Significant differences exist, however. In AUIML there is an implicit binding to the instance data as well as an explicit binding using the BINDING attribute, but there is no definition of how that binding works since that is determined by the runtime. XForms has an explicit definition of the instance data to which the user interface is bound. XForms also builds on XML Schema [6] for its data typing and augments it with XForms dynamic constraints. At the time AUIML was defined, there was no XML Schema and only DTDs were available. AUIML defined a small number of primitive data types that could be used. Although AUIML preceded XML Schema, its base types are very close to the corresponding built-in primitive types. AUIML also allowed dynamic constraints, properties such as the minimum and maximum value of a number that can be updated at run time. While there is no sophisticated dynamic constraint language defined (XForms uses XPath for this purpose), the application can access and update the constraints. In the AUIML toolkit the application and data model are written in Java, so a great deal of power is available to the AUIML application developer.

SUMMARY

By encoding the user and system interactions independent of the target platform, and standardizing these across devices, AUIML provides the following benefits.

Users benefit from a single, consistent user experience. Removing inconsistencies reduces learning time, not only between applications but across multiple devices as well. This, in turn, can enable accelerated deployment of new applications on existing and new devices.

Developers only need to write the user interaction once, instead of for each different device and form factor. This reduces development time, cost, and maintenance. Since applications work with a common data model, the application remains unaware of its current device.

Application developers can provide presentation preferences to leverage platform features that extend beyond the basic data model. This flexibility allows a developer to specify presentation features when they will improve the application's ease of use.

Since an application using the common data model is abstracted from its device, it remains flexible to move to new devices as they arrive in the future. Changes in user interface technology can be immediately leveraged by all AUIML applications as soon as a new renderer is created. Instead of *n*-number of applications that need to be migrated to new devices, only one renderer must be migrated for all applications to leverage a new device.

In addition, AUIML can also facilitate future GUI standards compliance. AUIML currently enforces user interface standards for all applications on a device. Future standards added to AUIML will automatically update all applications on a device to the new standard.

The interpretation of the AUIML XML by various device renderers provides AUIML applications with both platform and device independence.

ACKNOWLEDGMENTS

We thank the AUIML development team and customers for their work in evolving the platform as well as the team that worked on the Druid and PDML projects that influenced the design of AUIML.

REFERENCES

1. Defining User Interfaces in XML http://www.posc.org/notes/sep99/sep99_rm.pdf
2. AUIML IBM alphaWorks web site. Available at http://www.alphaworks.ibm.com/tech/auiml
3. W3C XForms WG http://www.w3.org/MarkUp/Forms/
4. W3C Device Independence WG http://www.w3.org/2001/di/
5. Dubinko, M., Klotz, L. L., Merrick, R., Raman, T. V.: XForms 1.0, W3C Recommendation 14. October 2003, http://www.w3.org/MarkUp/Forms/
6. XML Schema Part 2: Datatypes http://www.w3.org/TR/xmlschema-2/

Into the mangle: Software engineers run creases through a user interface metaphor

Simon Crowle
Department of Computing
Bournemouth University
Fern Barrow, Poole
Dorset, UK
scrowle@bournemouth.ac.uk

ABSTRACT
This paper presents some of the lessons learned from a software engineering case study that adopted a formal approach to the modeling, development and implementation of a user interface metaphor. The application of metaphorical concepts in human-computer interaction is wide spread but informed by only a few high-level design processes or abstractions. As a result, the integration of these concepts with other design views is often both weakly specified and only tacitly understood. In an attempt to better understand this design activity, this paper reports on the use of ISML (an XML-based GUI specification framework) within the context of a streaming media software engineering project. An analysis of the engineers' modeling and development experiences using the ISML framework reveals valuable insights regarding the use of some model-based design views and their realisation in an interactive application.

Author Keywords
metaphor, model-based user interface design, GUI specification

INTRODUCTION
Metaphors are frequently used in everyday language to enhance the comprehension of concepts where unfamiliar or parsimonious expressions are less effective. Whilst definitions of metaphor can be traced back to Aristotle [32], today the term is frequently explained using Richards' [31] characterisation of *vehicle* and *tenor*. In the context of human-computer interaction, this linguistic device is most famously synthesised using new modalities of communication in the *desktop metaphor*.

Making its first appearance on the Xerox Star system [33], the desktop metaphor presents objects and actions with which a user is already familiar to enhance their understanding of the system. During the course of interaction, both human and computer express or interpret the states of their respective domain objects through transformations in to and out of the metaphor. The utility and limitations of this graphical user interface (GUI) design are well known [29] and it remains a dominant feature in many modern GUIs today. The desktop does not delimit the extent of the application of metaphor however; case-studies [17] [12] and experimental work [4] [23] can be found that demonstrates the use of other vehicles. It is also important to recognise that metaphors are not applicable to all potential user interface design solutions and that they draw their own criticisms [15] [26].

In order to bring some discipline to the development of metaphors, a number of guidelines have been proposed [21] [1]. However, these recommendations are craft-based and provide little in the way of formalised development method - relatively little work exists in this area. Formal psychological [14] or mathematical models [18] provide interesting abstractions but these do not extend to support the activities of GUI design. A methodological framework is provided by Alty [3] that relates metaphor features to the system image. Coupled to this is a high-level, six-stage generative process that checks concepts against guidelines. Whilst this approach is likely to result in much more appropriate and effective concepts for a user interface design, it does not cast them into formal constructs that could be integrated with contemporary model-based design views. More recently, work by Crowle and Hole [10] [9] has provided an architecture that explicitly caters for metaphorical concepts and their synthesis with other aspects of graphical user interface design.

Arguably at present there is little support for formal, model-based design for user interface metaphors. But should anybody care? Is it not the case that most underlying windowing systems provide sufficient support for such concepts in the form of customisable icons and bitmaps? The answer is 'yes, we should' to the former question and 'no' to the latter if we consider that user interface metaphors:

- Transcend user interface technologies
- Travel across technology platforms
- Should have a meaningful relationship with a user's task knowledge
- Are pervasive, numerous and impact upon system design

It would be over simplistic to describe the ubiquitous desktop metaphor simply as the representation of system data as images, where previously they were displayed as text in a command line interface. The former explicitly re-casts the system domain into objects and actions that are familiar to the user whilst the later does not. Ironically, both of these interaction paradigms share a number of *metaphorical concepts* (such as files and folders) that were created for users but have no relation to the actual physical operations executed on their behalf by the underlying system. By the same token, these concepts may 'travel' across computer platforms to be realised in a variety of different ways both according to the extent to which a designer chooses to employ them and also the technological constraints of the solution. The point here is that the vehicles that are chosen to mediate the translation of a user's intentions into interactions extend beyond the limited scope of a command line mnemonic or any particular implementation of a GUI folder icon or button.

As a mediating device for human-computer interaction, the metaphor should provide meaningful mappings to a user's task or domain knowledge such that the semantic and articulatory distances [27] required to affect interaction are reduced. Simply overlaying iconographic imagery onto a GUI as an after-thought suggests only the weakest of relationships between the system and the user's world of tasks. In fact, a user interface metaphor is considerably more sophisticated and frequently exposes many partial mappings between concepts from both domains, some of which are used whilst others are not [3]. To understand where these congruities lie demands a more formal approach to the specification of a metaphor model and its relationship to both task and system abstractions.

Finally, not only do metaphors have a pervasive role in everyday language [20] but they also feature strongly in colloquial and formal computing discourse [19]. Indeed, the term 'software engineering' and many of the development activities associated with the practice are regarded to carry a substantial number definitions, concepts and analogies borrowed from other domains [7] [16]. In other words, metaphorical constructs will be employed by all stakeholders in a software project and their use is likely to have an impact on the way that ideas are communicated within the team as well as on the system design.

In the remainder of this paper a specification framework that has been designed to support metaphor abstractions is outlined and its application to a software engineering case-study examined. Section 2 introduces the Interface Specification Meta-Language [10] (ISML) and its background in model-based, user interface design. A brief background to the Urban Shout Cast (USC) project is given in section 3 and an overview of design and evaluation method is provided. Finally, section 4 reflects on the design behaviours identified in the project and some of the lessons learned from the study is discussed in section 5.

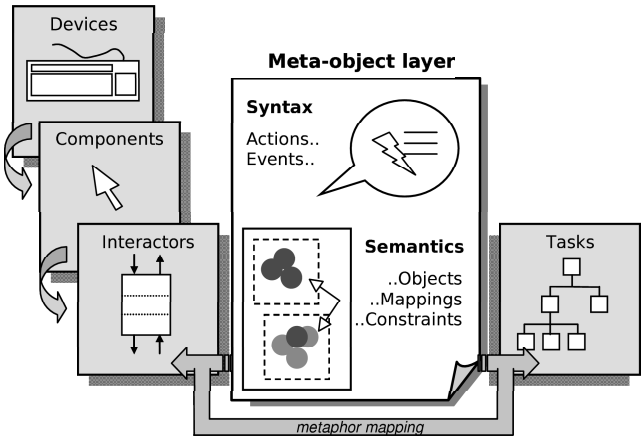


Figure 1. ISML Overview

MODEL-BASED METAPHOR SPECIFICATION

The model-based user interface design (MB-UID) paradigm seeks to provide abstractions, systematic methods and tools that will enhance current interactive software development practice [11]. A wide range of design views are considered within the scope of MB-UID including device, presentation, dialog, task and application-domain models [5] [22] [35]. An orthogonal diversity of underpinning formalisms and technologies also exists to describe, and in some cases provide tools support for, such design views - these include algebraic specification, state charts, petri-net and UML hybrids. Whilst the MB-UID community have yet to settle on a standard set of concepts and supporting technologies, there is an increasing movement toward XML-compliant specification of these models [34]. This is an important step forward since XSLT transforms allow models described in this way not only to be compared and potentially re-used in multiple MB-UID frameworks, but also translated into source code for rapid prototype generation.

ISML is a XML-based user interface specification language specifically developed to explicitly describe and map metaphor concepts to and from task and interactor abstractions. It is not the purpose of this paper to provide a comprehensive treatment of the ISML framework, however a brief overview will be given here (interested readers should see [10] [8]). Figure 1 depicts a simplified architectural view of ISML which consists of five principal layers that are connected by declarations of association, inheritance or various types of mapping. Physical and elementary logical definitions of user input and system output are specified in the devices and components layers. The former details particular capabilities of the interactive hardware whilst the later combines one or more devices together in a logical association to model a virtual device, such as a graphical pointer. Whilst both of these layers are important in defining the eventual realisation of the interface in terms of basic 'look and feel', they should be considered as peripheral to ISML's main views which are interactors, tasks and the principal abstraction: the meta-object layer.

The underpinning meta-object layer allows the designer to specify syntactic and semantic rules that later govern the declaration and behaviour of objects in both interactor and task layers. The rules, which determine types of attribute mapping, communication messages and simple behaviours are associated with objects that maintain an internal, abstract model. Within the ISML framework, these are the building blocks (declared in the *meta-object* layer) for modeling the world of tasks, the implemented GUI and the intermediary shared concepts that link the two: the metaphor. Interactors are essentially specialisations of the metaphorical objects and behaviours described in the meta-object layer; they invoke component presentations and link with the application model to present the user with an interactive realisation of the metaphor. The task layer uses the same meta-object definition scheme to describe task objects (and behaviours that they can enact). These are then referred to in a hierarchical decomposition of a user's task. Finally, the transformational aspect of the GUI metaphor described by Alty [2] is specified as mappings between the two meta-object based views; attributes, states and actions from the task world are linked (where possible) to those exposed by the interactor design.

```
<MetaphorMap>
<ObjectMap TaskObject="Task_DJ"
    TargetInteractor="Interactor_DJ"/>

<ObjectMap TaskObject="Task_PlayList"
    TargetInteractor="Interactor_PlayList"/>
...
<ActionMap
    TaskObject="Task_DJ"
    TaskAE="Task_Add"
    Qualifier="AFFECTS">

    <Implements TargetInteractor="Interactor_DJ"
    TargetAE="AE_Pick" Qualifier="AFFECTS"/>

    <Implements TargetInteractor="Interactor_DJ"
    TargetAE="AE_Drop" Qualifier="AFFECTS"/>

</ActionMap>
</MetaphorMap>
```

In the example above, elementary object mappings describe associations between entities specified in a task view (a 'DJ' and 'playlist') and their realised forms within the interactive system. The action map tags link a *task* action with two *metaphorical* actions ('pick' and 'drop') as they are expressed and implemented as *interactions* by the interactor DJ object (see section 3 for context of this example).

Whilst ISML has adopted many well established concepts from the MB-UID community, their synthesis and extension to explicitly express metaphorical concepts is new. Still very much in its infancy, this approach has much that needs close examination and validation. How well does ISML scale? Does it effectively and efficiently express design concepts? How much effort is required for ideas to reach implementation? How would its use impact on a software engineering project? These are just a few of the issues that challenge ISML as well as other model-based GUI design environments. If model-based methods are to support user interface design properly, it is important that their development is informed by empirical data generated from case-studies. As research within the MB-UID community matures, it is now possible to find examples of how such approaches are used to either model or implement interactive applications [24][6]. Reports like these disseminate, prescriptively, a demonstration of the utility of the MB-UID paradigm as practised by experts in the field. However, wide-spread acceptance of this approach requires that these concepts and tools are accessible, understandable, usable and perceived as relevant to the wider software engineering community. Integrating all these views is a hard problem to solve [30] and one that will demand, amongst other contributions, a perspective from the experiences gained from examining the *usability* of model-based, user interface design abstractions.

USC CASE STUDY DOMAIN

The *Urban Shout Cast* (USC) application was specifically devised as a software engineering project and case study for ISML. USC is a collaborative, shared environment in which physically separated users virtually meet (through networked PCs) to produce and broadcast an Internet radio show. The USC domain was chosen because it was considered as potentially 'metaphor rich' and sufficiently non-trivial to design. Users can log on as either a producer or as a DJ (who may assume any number of sub-roles, such a programme host or news reader). The producer manages DJs, their 'air time', the quality of the audio that listening clients receive as well as various other administrative responsibilities. DJs may log on, organise media, create 'play lists' (mini schedules outlining their programme) and transmit their part of the show to the producer.

Two independent teams of four final year undergraduate, software engineering students were given six months to develop a USC prototype based on the *functional* requirements provided by the author. Both teams had a basic understanding of HCI and user interface design principles and were advised that the project was to aim for novel, metaphor-rich features but that the actual design was under their own creative control. Access to the problem domain was made available to the teams through exposure to the working environment and users of their university's local radio station. Each team completed (and sometimes re-visited) four distinct design phases during the course of the project; these were: requirements definition; functional specification; design and prototype build. Throughout the project, both teams met separately with the author to discuss project issues; all meetings were recorded on mini-disc and subsequently transcribed. In the latter half of the project, all members of the USC groups were given a short tutorial of the ISML framework and subsequently participated in a series of elicitation meetings conducted by the author during which aspects of their design were interactively specified using the ISML framework.

Analysis

What does it mean to say that a model-based user interface design paradigm is 'usable'? Criteria and metrics for such an assertion remain undefined for the MB-UID community. Due to this vacuum and the relative novelty of ISML, it was decided that a two-tier approach to evaluation should take place: qualitative and model-analytical. Two broad direc-

tions for analysis were specified at the outset of the case-study:

1. What are the reactions of the developers to the use of ISML?
2. To what extent does ISML capture a design?

Dimension one was further sub-divided into two parts:

Development of the user interface metaphor How were the metaphorical concepts generated by the teams specified and refined over the course of the project? Did ISML help generate or modify ideas or did it confound them?

Perceptions of the design team What were the perceptions of utility and practicality of ISML? Where did the teams find ISML helpful and which areas where difficult or required a lot of effort to use?

Dimension two was similarly sub-divided:

What aspects of design does ISML capture? To what extent does the ISML framework adequately express the concepts and implementation details generated by the case study? What was missed?

Scope of metaphor abstraction Can ISML effectively abstract and map metaphor concepts to other design views? Is this mapping effective or could it be improved?

Dimension one

Analysis for dimension one was conducted using a qualitative analysis (based on the grounded-theory method [28]) of the interactive specification meetings in which design features were translated into ISML concepts. Meetings were formalised around the ISML framework and followed a specification methodology that is summarised in brief here:

1. Generate a hierarchical task model based on domain analysis.
2. Enumerate and associate task objects with task nodes.
3. Verify states, conditions and constraints associated with task execution.
4. Identify principal metaphorical objects that link tasks with system functionality
5. Identify actions associated with each object.
6. Identify mappings and constraints asserted by each object.
7. Specify input and output devices.
8. Specify high-level interactors that implement meta-objects.
9. Specify interactions that enact actions on objects.
10. Verify mappings between task, interactor and meta-object views.

Steps within this process where often iterative or recursive where complexity and the need for refinement was encountered. Upon project completion, each group was individually de-briefed on the case-study and asked to respond to a series of semi-structured questions designed to elicit the opinions and attitudes towards their experiences using ISML. Transcribed data from each groups' meetings (approximately 37-000 words each) was marked up by applying open and axial coding methods [13] using the analysis package *Atlas.ti* [25]. Concept groupings and semantic networks were generated for task, meta-object and interactor design views such that *inter* and *intra*-group comparisons could be evaluated.

Dimension two

The models elicited from each group provided the basis for a logical analysis of the ISML framework with respect to aspects of its expressiveness and effectiveness. In order to identify the 'capture capability' of the framework, two sources were used to make relative comparison: the concepts as they were discussed during design meetings (constructed in dimension one) and the executable prototype finally deployed by each team.

Critical evaluation of the design views was conducted in the same order as dimension one. For each view, the following assessments were made:

1. The extent to which ISML represented design parts (identified or missing)
2. The use of available ISML concepts in the expression of USC design
3. Missing or incomplete ISML data
4. Relative complexity of the model

Following this exercise, those parts from the abstracted metaphor model that both groups shared (a limited range) were unified by the author for the purpose of evaluating whether or not such a model could support both teams' interactor-based solution.

RESULTS

The scope of this paper does not allow a detailed treatment of the analysis (see [8]) but instead must restrict itself to a summary of the design behaviours uncovered.

Design behaviours

Five common behaviours were exhibited by both groups, these were:

Design reductionism The relative complexity and scale of the problem domain is progressively reduced as it was re-cast first from tasks to supporting metaphors and then again from metaphors to implementation details. In some cases, significant entities that appear in the task view are entirely removed from the interactor design.

Use of non-concrete concepts Whilst each group were capable of specifying physical objects (including their attributes, states and actions) within the ISML framework,

they encountered considerable difficulty in incorporating important but 'invisible' concepts. An example of this is the concept of a 'track' in a media object *or* collection; this is a conceptual entity that appears to be a part of many different physical objects but has no single, tangible presence.

Implementation bias Considerations of implementation design concepts and context impact significantly on the development and integration of the metaphor view and to a lesser extent the task view. In particular, references to structures such as 'files' and audio streaming formats appear frequently during all elicitation stages to pollute otherwise coherent domain descriptions.

Metaphor mangling As a consequence of the influences of the above design behaviours, surviving metaphors found in the eventual implementation became 'mangled'. This is to say that they frequently are only partial representations of a shared concept which become misplaced and/or distorted. A good example of this is the DJs' *playlist* - a list-like object of audio tracks which is used not only to represent a broadcast schedule but also a functional interaction point for playing music. The tracks themselves are disassociated not only from the media container they belong to (such as a CD or mini-disc) but also of a media playing device (such as a hi-fi).

Common model re-use There are a number of occasions when underlying meta-object layer structures are re-used to support common features or behaviours of similar metaphorical entities even though their interactor implementations are quite different. An example of this can be seen in the re-use of mixer desk sliders to enact modifications in the properties of audio components (a functional requirement).

Metaphor mangling shows most clearly that all five design behaviours potentially interact with one another to result in problematic or unusual design results. Not all of the outcomes are necessarily undesirable, however. For one of the groups, an unexpected synthesis of DJ mixer desk concepts with the common 'drag and drop' desktop interaction resulted in an interesting and novel audio *mixing and switching* mechanism (described later in this section).

ISML model features

A systematic comparison of ISML-encoded design views with both groups' executable prototypes and the qualitative concept models generated during the elicitation phase yielded insights into the successes and failures of the framework.

Devices and components Since most of the interaction deployed by each development team was simple, no significant problems were encountered in specifying the devices and presentation components required for each prototype. Unspecialised mouse and keyboard actions combined with simple or arrayed bitmap presentations represented the majority of the components used in this case.

Meta-object view A variety of hierarchical or compositional arrangements of meta-objects were developed to simulate the behaviour of core metaphorical concepts including DJs, media players and play lists. Both groups articulated a limited range of mappings, constraints and afferent action-events enacted by the DJ (a proxy for the user). Problems with this view arose in a severe lack of specification of abstract, non-concrete objects such as tracks and inter-DJ communication. There was evidence in a number of cases that the elicitation method failed to capture sufficient operational detail for objects - in particular the behaviour of some objects in response to efferent events. Both groups also showed redundancy in their object specification indicating the need for a strengthening of inheritance capabilities within ISML.

Interactor view It was possible for the USC engineers to specify basic relationships between their implementations and the meta-object abstractions; these were mostly one-to-one mappings but a few instances occurred in which interactors were aggregated to realise the underlying model. Neither of the groups were able to produce detailed *display* and *controller* part specifications for their interactor models. In their final implementation both teams, in varying degrees, fell back on common desktop components and metaphors in an attempt to manage complexity; as a result the effort required to specify these supporting interactors soared - this represents a major challenge facing ISML.

Task view Ironically, whilst the discussions of domain tasks were highly complex the actual models produced were comparatively impoverished. Both groups generated a hierarchical descriptions of core tasks including basic descriptions of domain objects and a small number of constraints for task execution. However, a number of task objects were not included - in particular those which were highly complex or abstract (the many different forms of inter-human communication within the broadcasting environment is a good example of this type of complexity).

Finally, having spent time on all the views supported by ISML, the teams were encouraged to map task actions and objects to interactors, and their the underlying metaphorical objects and actions. This exercise was somewhat simplistic and could have been refined further, however project time constraints did not allow further elaboration. Unification of the shared, core metaphor abstractions involved a process of compiling the most compatible and fully specified aspects of each teams' design - this resulted in a small but significant domain of objects, behaviours, mappings and constraints. Although reduced, this model was still substantial and space in this paper does not permit a complete review (see [8]). However, a snapshot of part of this generalised model can be seen in figure 2 in which both teams' use of meta-objects for placing a DJ 'on air' and adjusting their output volume are shown. The component parts of various interactors are highlighted and linked to meta-object summary boxes that depict attributes, mapping-constraints (MCs) and action-events (AEs). Mouse actions captured by the interactor model are translated into equivalent abstract actions upon objects (and result in a cascade of further communications, not shown) within the meta-object model.

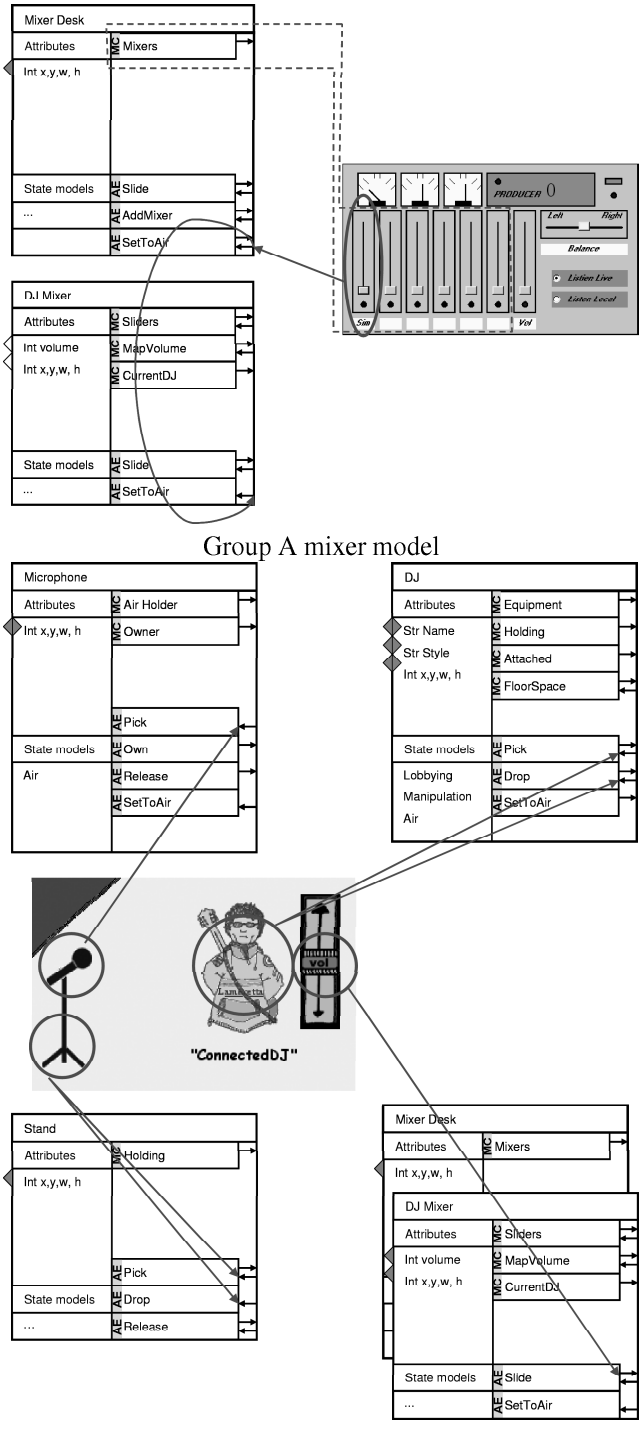


Figure 2. USC Mixer models

Group A's solution represents DJs and their on-air status through proxy mixer sliders on a mixer desk. In the unified model, these mixer sliders maintain a reference to the DJ object within a mapping-constraint set whilst the mixer desk acts as a manager adding or removing references as appropriate. In contrast, group B do not explicitly implement a mixer desk (although it still operates in the meta-object layer) but instead provide a DJ, microphone and stand. Although the microphone and stand objects are not shared by group A, they were included in the unified model because they were interesting and could be integrated without interfering with the design as a whole. The communication sequence for the same operation in group B's solution is longer since it initially involves a set of communications that essentially enact the placing of a DJ on air by giving him or her a microphone. However the same *'Set to Air'* sequence eventually reaches the mixer desk. Similarly, sliders may be dragged and dropped on to the appropriate DJ to modify their audio broadcasting properties.

DISCUSSION

The USC case study has generated insights into design behaviours and a review of ISML construct usage in a software engineering project. Clearly, this is useful for the further developments specific to ISML, but it also suggests important lessons for model-based, user interface design activities. In particular, it highlights the fact that whilst it may be possible to formally well define design views and their mappings, it may not necessarily be the case that they will be *used* in the same rigourous fashion during actual design. There may be many reasons for this; in the USC case-study time pressure and the novelty of a model-based GUI method certainly had an influence on the results. More interestingly however, the qualitative analysis shows that the developers were clearly influenced by other design problems and solution metaphors (such as media streaming formats and their choice of GUI application programmers' interface) whilst they were constructing their models. Though these biases may be benign (or sometimes even a source of engineering creativity) they demonstrate a reality of modeling practice: views impact on one another in unpredictable ways.

Additionally, the complexity of design views was variable in three senses. Firstly, the degree to which designers were willing to discuss the scope of the domain. During design meetings, task concepts were broad and complex but in comparison, potential implementation solutions were narrowly focused. Secondly, the elicited models did not always correspond with the concepts discussed - this is particularly true of the task view in which much of the richness of the radio broadcaster's world of work was discarded. Thirdly, the ISML framework itself imposed limitations of the expression of these views in that a) not all of the language's concepts were used and b) there were a number of design concepts that it could not adequately express (such as 'invisible' metaphorical concepts). All this makes specifying mappings complex; the specifier must acquire skills that will allow him or her to manage complexity and recognise the strengths and weaknesses of their modeling activities.

The proposal of even more development activities being inserted into their existing software engineering processes did not initially please the USC teams. For the software engineering community to accept new model-based methods for user interface design, they must be convinced of its effectiveness, relevance to their other activities and 'value for money' in terms of effort put in for products produced. Presently, ISML does not scale well. As well as issues related to the cost in specification effort importing already existing (and complex) interaction paradigms, the framework is itself large and generates very complex models (the unified USC model is over 100 pages of XML). Even though the teams were not actually writing the XML themselves, they considered the effort of specifying their design to be substantial.

USC Team reflections

Post-project, the teams were interviewed to gain some feedback from their experiences. Both teams felt that the process had got them to think in useful, new ways about their design issues, a member from group A described this as:

> "I think it makes you look at it [the design] in a different way ... as we develop it, you're thinking ... of implementation things at the same time, and that obviously incorporates it where as ... doing it this way, you might do things a bit differently."

The teams also highlighted some of the specification issues they found difficult. Group B directly sighted the problems they encountered with both non-concrete concepts and implementation bias:

> "Abstraction was very difficult... It was hard to separate the abstract thing from the actual.. But I think what complicated it more was the fact that we had what we thought we were going to implement and we had the ideal system ... it was was like pulling the two apart."

Despite this, there was a general, concluding consensus that with practice and deployment at early stages of the design process that these additional development activities would be useful and even creative. The effort required to produce ISML models was considered to be a problem however; both groups thought that "the documentation may be a bit too much" - the scalability issues identified in both analysis dimensions testify to this concern.

CONCLUSION

This paper has presented a summary of the findings from the case study that used a model-based specification framework for user interface design. The relative successes and failures of ISML were presented against the backdrop of five distinct and interacting design behaviours. These experiences indicate that the inclusion of MB-UID activities within the broader software engineering process is likely to result in a complex interaction of design practices. As a consequence, whilst it is important to continue to strive towards a rigourous and definitive MB-UID ontology, so to is it vital that such a framework is supported by the development of methods and tools that has been informed by research that ensures that they are effective and useable.

REFERENCES

1. D. Akoumianakis and C. Stephanidis. Multiple metaphor environments: Architectural premises for continuous interactions. In T. Turner, G. Szwillus, and M. Czerwinkski, editors, *"Continuity in Human Computer Interaction", Workshop in the context of the Conference on Human Factors in Computing Systems (CHI 2000 - The Future is Here),*, page 6, The Hague, The Netherlands, 2000. ACM Press.
2. J. L. Alty and R. P. Knott. Metaphor and human computer interaction: a model based approach. In C.L. Nehaniv, editor, *Proceedings of Computation for Metaphors, Analogy and Agents: An International Workshop*, pages 307–321. Springer-Verlag, 1999.
3. J. L. Alty, R. P. Knott, B. Anderson, and M. Smyth. A framework for engineering metaphor at the user interface. *Interacting with Computers*, 13(2):301–322, 2000.
4. W. Ark, D.C. Dryer, T. Selker, and S. Zhai. Representation matters: the effect of 3d objects and a spatial metaphor in a graphical user interface. In Hilary Johnson, Lawrence Nigay, and Chris Roast, editors, *Proceedings of HCI 98, the Conference on Human-Computer Interaction*, pages 209–219. Springer, 1998.
5. R. Bastide and P. Palanque. A visual and formal glue between application and interaction. *Journal of Visual Languages and Computing*, 10(5):481–507, 1999.
6. S. Berti and F. Paterno'. Model-based design of speech interfaces. In J. A. Jorge, N. J. Nunes, and J. F. Cunha, editors, *Interactive Systems Design Specification, and Verification : 10th International Workshop, DSV-IS 2003*, pages 231–244, Funchal, Madeira Island, Portugal, 2003. Springer.
7. A. Bryant. It's engineering jim ... but not as we know it. pages 78–87, Proceedings of the 22nd international conference on Software engineering, 2000. ACM Press.
8. S. Crowle. *The design and evaluation of a specification framework for user interface design*. Doctoral thesis, Bournemouth University, 2003.
9. S. Crowle and L. Hole. Seeing the wood for the trees: A framework for the specification of metaphor in interface design. In *Workshop on Integrating Multimedia, Metaphors and Multimodality , in PC-HCI 2001: Human Computer Interaction 2001*, pages 19–24, Patras, Greece, 2001. Typorama Publishers.
10. S. Crowle and L. Hole. An interface specification meta-language. In Joaquim A Jorge, Nuno J Nunes, and Joo F Cunha, editors, *DSV-IS 2003 : Issues in Designing New-generation Interactive Systems Proceedings of the Tenth Workshop on the Design, Specification and Verification of Interactive Systems*, pages 381–396, Funchal, Madeira Island, Portugal, 2003. Springer.

11. P. P. da Silva. User interface declarative models and development environments: A survey. In P. Palanque and F. Patern, editors, *Interactive Systems. Design, Specification, and Verification, 7th International Workshop, DSV-IS 2000*, pages 207–226, Limerick, Ireland, 2000. Springer.

12. A. Dieberger and A. U. Frank. A city metaphor to support navigation in complex information spaces. *Journal of Visual Languages and Computing*, 9(6):597–622, 1998.

13. N.G. Fielding and R.M. Lee, editors. *Computer Analysis and Qualitative Research*. Sage Publications, 1998.

14. D. Gentner, B. Bowdle, P. Wolff, and C. Boronat. Metaphor is like analogy. In D. Gentner, K. J. Holyoak, and B. N. Kokinov, editors, *The analogical mind: Perspectives from cognitive science*, pages 199–253. MIT Press, Cambridge, MA, 2001.

15. F. Halasz and T. Moran. Analogy considered harmful. In *Human Factors in Computer Systems Conference, 1982*, pages 383–386. NBS, 1982.

16. J.D. Herbsleb. Metaphorical representation in collaborative software engineering. In *Proceedings of the international joint conference on Work activities coordination and collaboration*, pages 117–126, San Francisco, California, US, 1999. ACM Press.

17. L. Hole, S. Crowle, and N. Millard. The motivational user interface. In J. May, J. Siddiqi, and J. Wilkinson, editors, *Human-Computer Interaction'98*, pages 68–69, Sheffield Hallam University, UK, 1998.

18. B. Indurkhya. Constrained semantic transference - a formal theory of metaphors. *Synthese*, 68(3):515–551, 1986.

19. G.J. Johnson. Of metaphor and the difficulty of computer discourse. In *Communications of the ACM*, volume 37, pages 92–102, New York, USA, 1994. ACM.

20. G. Lakoff. The contemporary theory of metaphor. In Andrew Ortony, editor, *Metaphor and Thought*, pages 202–251. Cambridge University Press, 1992.

21. J. Lovgren. How to choose good metaphors. *Ieee Software*, 11(3):86–88, 1994.

22. K. Luyten, T. Clerckx, K. Coninx, and J. Vanderdonckt. Derivation of a dialog model from a task model by activity chain extraction. In J. A. Jorge, N. J. Nunes, and J. F. Cunha, editors, *Interactive Systems Design Specification, and Verification : 10th International Workshop, DSV-IS 2003*, pages 203–217, Funchal, Madeira Island, Portugal, 2003. Springer.

23. P. Maglio and T. Matlock. Metaphors we surf the web by. In *Workshop on Personalized and Social Navigation in Information Space*, pages 138–149, Stockholm, Sweden, 1998.

24. P. Markopoulos, P. Shrubsole, and J. de Vet. Refinement of the pac model for the component-based design and specification of television based interfaces. In D.J. Duke and A. Puerta, editors, *6th International Eurographics Workshop on Design, Specification and Verification of Interactive Systems*, pages 117–132, Braga, Portugal, 1999. Springer.

25. T. Muhr. Atlas.ti, 2002.

26. B. Nardi and C. Zarmer. Beyond models and metaphors: Visual formalisms in user interface design. *Journal of Visual Languages and Computing*, 4:5–33, 1993.

27. Donald A. Norman and Stephen W. Draper. Cognitive engineering. In Donald A. Norman and Stephen W. Draper, editors, *User Centred System Design*, volume 1, pages 31 – 61. Lawrence Erlbaum Associates, 1986.

28. N. Pidgeon. Grounded theory: theoretical background. In J. Richardson, editor, *Handbook of Qualitative Research Methods for Psychology and the Social Sciences*, pages 75–85. Kogan Page Ltd, 1996.

29. J. Preece, Y. Rogers, H. Sharp, D. Benyon, S. Holland, and T. Carey. *Human-Computer Interaction*. Addison-Wesley, 1994.

30. A. Puerta and J. Eisenstein. Towards a general computational framework for model-based interface development systems. *Knowledge-Based Systems*, 12(8):433–442, 1999.

31. I.A. Richards. *The Philosophy of Rhetoric*. Oxford University Press, 1936.

32. P. Ricoeur. *The Rule of Metaphor: Multi-disciplinary Studies of the Creation of Meaning in Language*. Taylor and Francis Brooks Ltd, 1986.

33. D. C. Smith, C. Irby, R. Kimball, B. Verplank, and E. Harslem. Designing the star user interface. *Byte*, 7(4):242–282, 1982.

34. N. Souchon and J. Vanderdonckt. A review of xml-compliant user interface design description languages. In J. A. Jorge, N. J. Nunes, and J. F. Cunha, editors, *Interactive Systems Design Specification, and Verification : 10th International Workshop, DSV-IS 2003*, pages 377–391, Funchal, Madeira Island, Portugal, 2003. Springer.

35. H. Traetteberg. Dialog modelling with interactors and uml statecharts - a hybrid approach. In J. A. Jorge, N. J. Nunes, and J. F. Cunha, editors, *Interactive Systems Design Specification, and Verification : 10th International Workshop, DSV-IS 2003*, pages 346–361, Funchal, Madeira Island, Portugal, 2003. Spinger.

USIXML: A User Interface Description Language for Context-Sensitive User Interfaces

Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, Murielle Florins, and Daniela Trevisan
Université catholique de Louvain (UCL) – School of Management (IAG)
Information Systems Research Unit (ISYS) – Belgian Lab. of Computer-Human Interaction (BCHI)
B-1348 Louvain-la-Neuve, Belgium
Phone: +32-10/478525 – Fax : +32-10/478324
{limbourg, vanderdonckt, michotte, bouillon, florins, trevisan}@isys.ucl.ac.be

ABSTRACT
This paper presents USIXML (USer Interface eXtensible Markup Language), a User Interface Description Language aimed at describing user interfaces with various levels of details and abstractions, depending on the context of use. USIXML supports a family of user interfaces such as, but not limited to: device-independent, platform-independent, modality independent, and ultimately context-independent. This paper consequently details how context-sensitive user interfaces may be specified and produced from the USIXML specifications. USIXML allows specifying multiple models involved in user interface 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: task & concepts, abstract user interface, concrete user interface, and final user interface. To support relationships between these models, a model for inter-model mapping is also introduced that cover forward and reverse engineering as well as translation from one context of use to another.

Categories and Subject Descriptors
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.2.4 [**Database Management**]: Systems – *transaction processing*. I.3.6 [**Computer Graphics**] Methodology and Techniques – i*nteraction techniques.*

General Terms: Design, Languages, Human Factors.

Keywords: Computing platform, context-aware adaptation, device independent, model-based approach, multi-platform, user interface description language, user interface engineering.

1. INTRODUCTION
Since the dawn of the discipline of Human-Computer Interaction (HCI), people have attempted to define many languages addressing different aspects of user interfaces (UIs). In particular, a lot of effort has been devoted to introduce various UI Description Languages (UIDLs) with various objectives in mind:

- To introduce a UIDL as a specification language.
- To introduce a UIDL as a communication format.
- To introduce a UIDL to express portability in virtual toolkits.
- To support adaptation.
- To support computing-platform independence.
- To support context-sensitivity.
- To support different families of UIs such as multimodal UIs, multimedia, hypermedia, etc.

The main goal of this paper is to define a UIDL that cumulates the previous requirements into one single language. Such UIDLs may pursue various goals:

- Ensuring portability of UIs from one computing platform to another while preserving some consistency between [4] or with the target computing platform [1,18].
- Capturing UI requirements for an abstract definition that remains stable over time [1,5,19].
- Improving the reusability of UI design [11,25].
- Making one UI design for multiple devices, platforms, or appliances. This goal is often referred to as the device-, platform- , or appliance-independence rendering [1,3,8,9,15,16].
- Supporting extensibility and adaptability of UI [10].
- Using a UI description to enable automated generation of UI code [1,3,5,8,9,10,11,15,16,18,19,25].

On top of these goals are added to more goals that are considered uncovered by ongoing initiatives:

- Making one UI design independently of any modality of interaction (e.g., graphical UI, vocal UI, virtual UI, multimodal,…) so that a design a this level may initiate more concrete designs once a particular modality has been selected.
- Supporting the integration of any model used in the UI development process, such as, but not limited to: the context of use, the user, the platform, the environment, the devices used,…
- Expressing explicitly mappings between models and elements when appropriate to address the mapping problem [8].
- Supporting the continuous and seamless manipulation of models from the abstract level to the concrete levels, such as in the Model-Driven Architecture (MDA) of the OMG Group.
- Expressing UIs at a given instant of usage so as to capture relevant information to ensure runtime migration.

Reaching a UIDL that fully addresses all these requirements and encompasses all the properties of interest of all these types of UIs is certainly neither possible nor desirable. Therefore, this UIDL pursues the goal of capturing the essential properties of interest that turn out to be vital for specifying, describing, designing, and developing such UIs. Consequently, this paper will present and motivate the choices that have been made to drive the definition of USIXML, a UIDL addressing the above requirements by defining models involved in this process. The remainder of this paper is structured as follows: section 2 provides a state of the art in the domain of UIDLs addressing partially or totally the above requirements. Section 3 presents the structure of the USIXML language by showing its scope that is wider than some existing UIDLs. Section 4 defines the different models and mappings that constitute USIXML. The original part of USIXML is emphasized when appropriate. Section 5 concludes the paper by bringing up the main benefits of USIXML with respect to other UIDLs.

2. RELATED WORK

It is worth to notice that many initiatives addressing the design of UIs for multiple platforms almost resuscitated the problem of UIDL that was for a time left forgotten after the question of portable UI has been achieved. Consequently, many initiatives for solving the design of UIs for multiple computing platforms or multiple contexts of use simultaneously consider a UIDL and software based on this UIDL to produce various types of UIs.

The PIMA Project [11] aims at producing applications that are device independent. A Platform Independent Application can be created either by a design tool or by abstracting a concrete UI thanks to the generalization process. Generalization is done by reverse engineering [6] the code of the UI. This process starts with the detection of interaction elements. Secondly, the properties and semantic information of these elements can be inferred. A specialized engine with a device profile then creates another application specialized for a particular device.

TERESA [18] produces different UIs for multiple computing platform from a general task model which is progressively refined for the different platforms. Then, various presentation and dialogues techniques are used to map the general specifications expressed into XHTML code for each platform such as web, PocketPC, and mobile phones. The TERESA (Transformation Environment for inteRactivE Systems representAtions) exploits a UIDL called TERESAXML that supports several types of transformations such as: task model into presentation task sets, task model into abstract UI, abstract UI to concrete UI, generation of the final UI. In [25], a very interesting algorithm is provided that maps a hierarchical task model to a presentation model that explicitly takes into account platform characteristics such as screen resolution.

UIML [1] is also a UIDL intended to support the development of UIs for multiple computing platforms by introducing a description that is platform-independent that will be further expanded with peers once a target platform has been chosen. Recently, the TIDE tool introduced transformations from a basic task model.

XIML [8,19] is a UIDL containing mechanisms for specifying any type of model, of model element, and relationships between. Although some predefined models and relationships exist, one can expand the existing set to fit a particular context of use. XIML has been used in MANNA for platform adaptation [8], in VAQUITA to support reverse engineering [3], and in Envir3D to transform a graphical UI into a virtual Ui thanks to mapping tables [3].

WSXL [9] covers UI and Web services that are attached to these UIs. eNode (http://www.enode.com) is also an interesting UIDL in the sense that the dialog part is precisely described, especially at the widget level where abstract events allow a precise definition of any widget behavior. SeescoaXML [21] is the base UIDL exploited in the Seescoa project to support the production of UIs for multiple platforms and the run-time migration of the full UI at run-time. From the same specifications, multiple UIs can be generated for multiple computing platforms.

Many UIDLs exist today that cover similar and different requirements, some of them having been reviewed in [20]. USIXML, the UIDL that is presented in this paper, is similar to the above UIDLs in the sense that is also cover multiples aspects supported by these languages, but it is especially intended for context-sensitive UIs.

3. STRUCTURE OF USIXML

USIXML is structured according to the four basic levels of abstractions defined in the Cameleon reference framework [4] that is intended to express the UI development life cycle for context-sensitive interactive applications (Fig. 1).

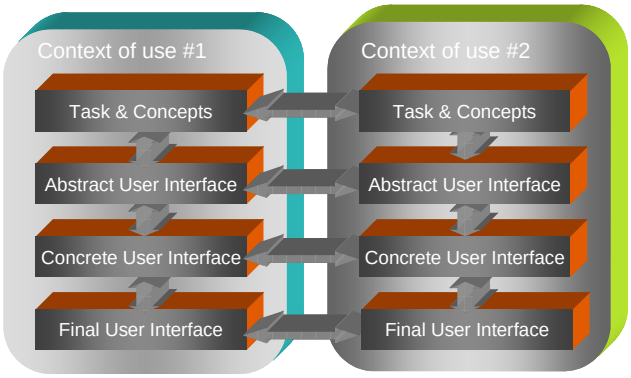


Figure 1. The four basic levels of the Cameleon reference framework [8].

At the bottom is the *Final User Interface* (FUI) related to any UI running on a particular computing platform either by interpretation (e.g., through a Web browser) or by execution (e.g., after compilation of code in an interactive development environment).

A *Concrete User Interface* (CUI) abstracts a FUI into a UI definition that is independent of any computing platform. Although a CUI makes explicit the final look and feel of a FUI, it is still a mockup that runs only within a particular environment. A CUI can also be considered as a reification of a AUI at the upper level and an abstraction of the FUI with respect to the platform.

An *Abstract User Interface* (AUI) abstract a CUI into a UI definition that is independent of any modality of interaction (e.g., graphical interaction, vocal interaction, speech synthesis and recognition, video-based interaction, virtual, augmented or mixed reality). An AUI can also be considered as a canonical expression of the rendering of the domain concepts and tasks in a way that is independent from any modality of interaction. For example, in SEGUIA [24], an AUI is a collection of related presentation units. The relations between the presentation units are inferred from the task relationships expressed at the upper level (task and concepts). An AUI is considered as an abstraction of a CUI with respect to modality.

At the top of the framework is the *Task & Concepts* level where the interactive task to be carried out by the end user is defined according to her viewpoint, along with the various objects that are manipulated by these tasks. These objects are considered as instances of classes representing the concepts manipulated.

A *transient model* [4] is an intermediate model that is required only momentarily during the development life cycle of a FUI. Task-oriented descriptions, AUI and CUI are typical examples of transient models.

On the other hand, *ontological models* [4] are meta-models that are independent of any domain of human activity (e.g., medical domain, surgery, and accounting) and any interactive system. Roughly speaking, they identify key dimensions for addressing a given UI design problem. When instantiated, they give rise to *archetypal models* [4] that are models dependent of an interactive system for a given domain of human activity.

There are three ontological models for context-sensitivity [4]:

- Domain models that support the description of the concepts and user tasks relative to a domain;
- Context models that characterize the context of use in terms of user, platform, and environment. The context model is consequently further decomposed into a user model, a platform model, and an environment model. At least one of these sub-models should be present to build a context model.
- Adaptation models that specify how a UI can be adapted after a change of the context of use that is significant enough to trigger

4. CONTENTS OF USIXML

4.1 Task

A *task model* describes the various tasks to be carried out by a user in interaction with an interactive system. After a comparison of several task modeling technique, an extended version of ConcurTaskTree (CTT) [17] has been chosen to represent user's tasks and their logical and temporal ordering. A task model is therefore composed of *tasks* and *task relationships*.

Tasks are, notably, described with a name, a type (user's, interactive, system and abstract [17]), a task frequency (relative frequency of execution of a task. *Task frequency* is evaluated on a scale from 1 to 5. A value of 1 meaning that a task has a low frequency, 5 meaning that a task is very frequent), a task importance (relative importance of a task with respect to main user's goals. As task frequency, task importance is evaluated on a scale from 1 to 5. A value of 1 means that a task has a low frequency, 5 means that a task is very frequent). Frequency and importance are interesting attributes when it comes to adapt a UI to a constraining context imposing a UI system to be pruned. Finally, an *action type* is based on a taxonomy introduced to better qualify tasks the leave of a task tree. This taxonomy, strongly inspired by [12] (Table 1), is twofold: a verb describes the type of activity at hand; an expression designates the type of object on which the action is operated. By combining these two dimensions a fine derivation of interaction objects supposed to support a task becomes possible.

Task relationships are of two main types: *decomposition* enables to represent the hierarchical structure a task tree hierarchical structure, *temporal* allows specifying a temporal relationship between sibling tasks of a task tree. LOTOS operators are used here.

Action	Item
Start/go, stop/exit, select, choose, create, delete, modify, move, duplicate, toggle, view, monitor	Operation, container, collection, element

Table 1. Taxonomy of action types for tasks.

4.2 Domain

A *domain model* describes the real-world concepts and their interactions as understood by users. Many formalisms have been introduced to represent systems of concepts: frames, semantic networks, entity relationship schemas, class diagrams,… USIXML domain model has the form of a UML class diagram. Concepts contained in USIXML domain model are at a certain point manipulated by users. By manipulated, it is meant that either attribute values are rendered through the UI or that methods attached to classes of objects are used by a user (i.e., triggered by a user event).

Domain model concepts are classes, attributes, methods and domain relationships. A *class* describes the characteristics of a set of objects sharing a set of common properties. A class may contain several attributes and methods. *Attributes* are described with their type, cardinality. Extensive specification of enumerated domains is possible. An original typology allows to characterize the type of domain of an attribute. Indeed, `attribute_domain_characterization` takes the value of: interval, continuous interval, discrete interval, linear interval, circular interval, set[*n*] (where *n* is the number of possible values in an attribute domain). Used in combination with a task model, this typology helps to map domain attributes to a type of interaction object by which it will be rendered. For instance, a choose element task on an attribute with a circular interval enable the derivation of a (multi-state) toggle button. *Methods* are described with their signature i.e., with their name, type, and parameters. A set of predefined method name inspired from OO patterns are used to facilitate the definition of generic design heuristics. For instance, the CRUD pattern is used any method realizing a Create, Read, Update or Delete operation [12]. Finally, *domain relationships* describe various types of relationships between classes. They can be classified in three types: *generalization*, *aggregation*, and *ad hoc*. Class relationships are described with several attributes enabling the specification of role names and cardinalities.

4.3 Context model

A *context model* describes all the entities that may influence carrying out the interactive task of user with the intended UI. It is assumed to capture any relevant attribute of the context of use, in which the user is. A context model consists of:

- A *user model* that recursively decomposes the user population into stereotypes (or profiles) and sub-stereotypes, each stereotype sharing a same series of attributes and associated values.
- A *platform model* captures relevant attributes for each couple software-hardware platform that may significantly influence the context-sensitivity. For instance, screen resolution of the platform is a major property taken into account in adaptation [8] and graceful degradation of UIs [10] when the UI designed for a normal screen is reduced for a more constrained screen. An interesting initiative related to platform modelling is the W3C CC/PP profiles (Composite Capabilities/Preferences Profile). A CC/PP profile is a description of device capabilities and user preferences that can be used to guide the adaptation of content presented to that device. Although CC/PP is not a vocabulary that would permit us to describe a platform, it is a generic XML-based language that allows to write vocabularies peculiar to various platforms. For the purpose of USIXML, we integrated a subset of CC/PP into platform families that are recursively decomposed.
- An *environment model* describes any property of interest of the physical environment where the user is using the UI on the computing platform to accomplish her interactive tasks. Such attributes may be physical (e.g., lighting conditions), psychological (e.g., level of stress), and organization (e.g., location and role definition in the organization chart).

4.4 Abstract User Interface (AUI)

A *AUI model* is a UI model that represents a canonical expression of the renderings and manipulation of the domain concepts and functions in a way that is independent from any modality and computing platform. An AUI is populated by *abstract interaction*

objects and *abstract user interface relationship.*

4.4.1 Abstract Interaction Object (AIO)

An AIO consists of any element populating an *AUI model* consisting in an abstraction of widgets found in most toolkits like windows, buttons but, also, vocal output widget in auditory interface. An AIO is supposed to be independent of any modality of interaction and any platform. AUI types are presented in hierarchy. The more a specification is precise the more the mapping to a concrete object will be precise. AIO are composed of multiple facets. We call them *multi-faceted***.** Each facet describes a particular function an AIO may assume. Four main facets are identified:

1. An **input facet** describes the input type accepted by an AIO.
2. An **output facet** describes what data may be presented to the user by an AIO.
3. A **navigation facet** describes the possible container transition a particular AIO may enable.
4. A **control facet** describes possible methods of the functional core that may be triggered from a particular widget.

An AIO may assume several facets in the same time. For instance, an AIO may display an output while accepting an input from a user, ensure a transition between windows and trigger a method from the functional core.

In order to group AIOs together, the *Interaction Space* is a type of *AIO* that support the execution of a set of logically/semantically connected tasks. An interaction space contains other interaction or other AIO's (see *grouping relationship).* It may be reified into one or more graphical containers like windows, dialog boxes or time slot in the case of auditory user interfaces. It is very important to note that an interaction space in not necessarily reified into a visible object. For instance, an *outputer* (a textbox) and *inputer* (a label) may be grouped together, their possible materialisations i.e., respectively a textbox and an inputer will only be bound together by an internal constraint (not perceivable by the user as is).

4.4.2 Abstract User Interface Relationship (AUI relationship)

An *AUI relationship* is an abstract relationship among *AUI objects* that indicate the existence of some spatio-temporal setting among them (e.g., a navigation between two interaction spaces). Relationships may have multiple source and multiple targets. Two main types of AUI relationships are therefore distinguished: dialog transitions and spatio-temporal relationships.

Dialog Transition is a type of *AUI relationship* that enables to specify a navigation transition between one interaction space and on another or several others with the following possibilities:

- **Suspend**: is a type of *AUI relationship* that enables to specify that the source interaction space is suspended to enable the target interaction space. A (reverse) resume relationship between these interaction spaces must exist for the coherence of the model.
- **Resume**: is a type of *AUI relationship* that enables to specify that the target window is re-enabled after having been suspended by a prior *suspend* relationship.
- **Disables**: is a type of *AUI relationship* that enables to specify that the source interaction space disables the target interaction spaces.
- **Enables**: is a type of *AUI relationship* that enables to specify that the source interaction space enables the target interaction spaces.

Grouping is a type of *AUI relationship* that enables to specify a collection of grouped *AIOs.* The source of a grouping relationship is always an interaction space. Additional information can be specified to precise the nature of the grouping relationship. For instance some ordering may be specified between grouped elements. This ordering can be based on an alphabetical order or a numerical order. Furthermore, it may be specified that grouped element must be specifically differentiated with each other (e.g., by using different colours or dissimilar tone of voices).

At the AUI level, the designer is interested in expressing only high-level relationships between AIOs, if any, without expressing low-level details of the relationships, such as specific distance or time. **Spatio-temporal relationships** characterise the physical constraints between AIOs as they are presented in time and space. Since an AUI does not preclude the usage of any particular modality, we do not know whether a particular AUI will be further reified into a CUI that is graphical, vocal, multimodal, or virtual. Therefore, spatio-temporal relationships should be expressed in a way that is independent of any modality.

For this purpose, the thirteen possible temporal relationships from Allen are considered. Basically, there are two types of temporal relationships (Table 2): sequential (*before* relationship) and simultaneous (that can be *equal*, *meets*, *overlaps*, *during*, *starts*, or *finishes* relationships). Each basic relationship has an inverse relationship, except the *equal* relationship which is symmetric. Although Allen relationships have been introduced to characterise temporal intervals, they are suitable for expressing constraints for space and time thanks to a space-time value. All simultaneous relationships (such as *overlaps*, *during*, *starts*, and *finishes)* can be generalised a the *equal* relationship by inserting some delay time when needed. For example, in the *x before y* relationship, there is a space-time value greater than zero between *x* and *y* while in the *x meets y* relationship the space-time value is equal zero between *x* and *y*. As relationships are abstract at the AUI level, the space-time value is left unspecified until needed at the CUI level. The spatial relationship between *A* and *B* is defined as follows:

Spatial_Composition (*A,B*) = (*Ri* , *Rj)*, where i, j ∈ {1,…,13}, *Ri* is the identifier of the spatial relationship between *A* and *B* according to the *X* axis and *Rj* is the identifier of the spatial relationship between *A* and *B* according to the *Y* axis in the matrix reproduced in Fig. 3. When a spatial arrangement is expressed only according to one dimension, *Ri* = ∅ ou *Rj* = ∅.

The temporal relationship between the *A* and *B* is defined as follows: *Temporal_Composition (A,B)* = (*Ri* , *Rj*), where i, j ∈ {1,…,13} as defined in Fig. 3.

4.5 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 lie 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 program UIs.
  - Concrete Interaction Objects 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). Orthogonally to AIOs, CIOs are devided into two types graphicalContainers (e.g., window, panel, table, cell, dialog box,…) and graphicalIndividualComponents (e.g., a button, a text component, an video component, a menu, a spin button,…). In SEGUIA [38], a CUI consists of a hierarchy of CIOs resulting from a transformation of AIOs [37].

- 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. 6 shows a simple declaration of a window containing a top-centered label and an OK button.

```
<window id="W1" name="Main Window">
      <box … type = "main" splittable=true detachable=false… >
      <box … type = "horizontal" >
      <textComponent id="TX1" name="Text1" offsetVertical="top" offsetHorizontal="center" defaultContent="Hello World!"/>
      </box>
      <box type="horizontal">
      <button id="B1" name="OkButton" defaultContent="OK" />
      </box>
      </box>
</window>
```

Figure 6. USIXML specification of a window containing widgets.

A CUI is equipped with a mechanism, called dialog, allowing the specification of the dynamic behavior of a concrete user interface. This mechanism covers a navigation definition language and a powerful event/action language. For clarity purpose, we isolated explanations on this aspect. To better understand the differences that exist between AIOs and CIOs in the context of USIXML, Fig. 2 shows that the FUI level is populated by the true final widget in the target platform, e.g., a Download pushbutton written in HTML and rendered on a MacOS X platform (bottom left of Fig. 2). At the FUI level, the HTML source code of this button may remain the same, but can be rendered differently depending on the browser, the platform and other parameters. At the CIO level, the different physical widgets are abstracted from their platform and classified into CIO types. At the AIO level, these objects are successively abstracted from their modality of interaction.

Fig. 2 shows that thanks to these different levels, it is possible to find out alternate CIO or AIO in case of change of the context of use, especially in graceful degradation [16]. Fig. 7 shows alternate CIOs for a menu AIO in the same graphical modality, while Fig. 8 shows alternate AIOs with different modalities.

4.6 Inter-model Mapping

Model integration is a well-known issue in model-based approach of UI development. Rather than proposing a collection of unrelated models and model elements, USIXML provides the designer with a set of pre-defined relationships allowing to map elements from heterogeneous models. This may be useful, for instance, for architecture derivation (mappings between domain and CUI/AUI models), for traceability in the development cycle (reification, abstraction and translation), for addressing context sensitive issues, for improving the preciseness of model derivation heuristics. The mappings between the different models are of several types:

- *Manipulates* maps a task onto a domain concept i.e., a class, an attribute, an operation or any combination of these types. This relationship has an attribute 'centrality' which specifies the relative importance of a domain element to the execution of its corresponding task. This item is evaluated on a scale of 1 to 5. 1 meaning that domain concepts is not central, 5 that is completely necessary (i.e., essential to the execution of the task).
- *Is Rendered By* maps a domain concept onto a presentation element either that a domain concept is subject to user input or that it is only presented to a user. An attribute of this relationship specifies if the values of the mapped attribute may be updated from the UI or not. If not, values are only visualized.
- *Is Executed In* maps a task onto an AUI or CUI element. It indicates that a task is performed through this (set of) AUI(s) or CUI(s) element(s).
- *Is Abstracted Into* and *Is Reified Into* map AUI and CUI elements. This relationship indicates that an element has been derived, through reification or abstraction (see framework of Fig. 1), from another.
- *Has Context* maps any model element to one or several contexts of use.
- *Corresponds To* maps a task temporal relationship with a navigation relationship as defined in a AUI or a CUI.

4.7 Dynamic aspects in USIXML

Dynamics of USIXML cover several aspects:

- **Requirements derivation (**dubbed **reification)** along our development cycle structure (Fig. XX). By requirement derivation it is meant the changing or translation of a high-level requirement into a form that is appropriate for low-level analysis or design.
- **Reverse engineering (**dubbed **abstraction)** along our development cycle (see Fig. 1). By reverse engineering it is meant the extraction of high-level requirement from a set of low-level requirements artifacts or from code.
- **Context (of use) adaptation (**dubbed **translation).** The context of use is defined as a triple of the form *(e, p, u)* where *e* is an possible or actual environments considered for a software system, *p* is a possible or actual target platform, *u* is a user category. Context adaptation is a process of modifying a user interface in consequence of a change of one or several element of the triple described above.
- **Dialog specification** of the user interface. Dialog can be defined as a description of user interface state change along with the event/action specification resulting in a state changes.

The three first items are referred with the generic term of **transformation**. Both transformation and dialog are specified using transformation systems. Transformation systems rely on the theory of graph grammars [20]. We first explain what a transformation system is. Transformations and dialog specification are, then, further explained.

4.7.1 Transformation systems

The proposed formalism to represent model transformation and dialog in USIXML is graph transformation. This formalism has been discussed in [19]). USIXML has been designed with an underlying graph structure. Consequently any graph transformation rule can be applied to a USIXML specification. This formalism conveniently applies to model transformation and dialog specification.

A transformation system is composed of several transformation rules. Technically, a rule is graph rewriting rule equipped with negative application conditions and attribute conditions [20].

Fig. 9 illustrates how a transformation system applies to a USIXML specification: let G be a USIXML specification, when 1) a Left Hand Side (LHS) matches into G and 2) a Negative Appplication Condition (NAC) does not matches into G (note that several NAC may be associated with a rule) 3) the LHS is replaced by a Right Hand Side (RHS). G is resultantly transformed into G', a resultant USIXML specification. All elements of G not covered by the match are considered as unchanged. To add to the expressive power of transformation rules, variables may be associated to attributes within a LHS. An expression may compare this variable with a constant or with another variable. This mechanism is called 'attribute condition'.

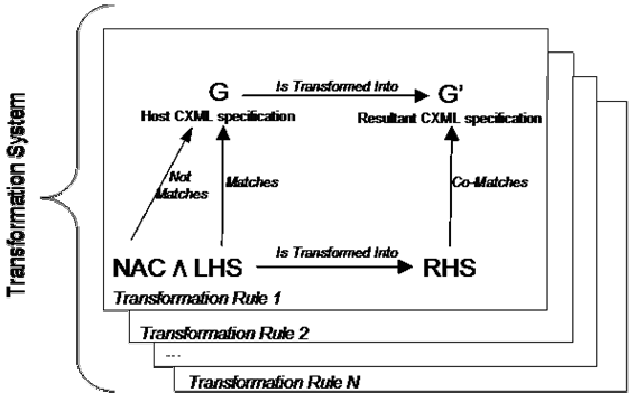


Figure 9. Characterisation of a transformation in USIXML.

4.7.2 Dialog specification

USIXML is equipped with a concrete dialog model. This dialog model is integrated into the concrete user interface model. For illustration purpose we have isolated the dialog parts from the CUI model. The basis of our dialog is an event/action language. Every concrete interaction object may be associated with one or several behavior specification. A behavior is a couple event/action.

- Events may be composite (composed by other events) allowing the expression of complex expressions.
  - Events may be chosen within a predefined event language (Table 3).
  - Events may be composed with temporal operators. Each operator is represented with its symbol: >> is a sequence, ||| is order independence, OR is a disjunction, (n) is an iteration where n is the iteration factor.
- Actions are performed by transformation systems. Transformation systems are sets of transformation rules operating on a specification. Actions have the expressive power of graph grammars. Graph grammars have been proved very powerful (as powerful as Petri nets) represent the behavior of dynamic systems. Concretely, the result of an action may be any change in the CIO model including the triggering of methods from the domain model. An example is given in Fig. 10.

CIO	Events
All graphical CIOs	movePointer(X,device), pointerOver(X,device), moveOutPointer(X,device), click(X,device), doubleClick(X,device), depress(X,device), release (X,device), dragOver(X,Y,device), dragDrop(X, Y,device), hasFocus(X), lostFocus(X)
graphicalContainer	resize(xFactor,yFactor)
textComponent	change
slider	move(cursor,x)
spin	spinUp, spinDown

Table 3. Events of CIOs.

```
<button …name="ClearButton1"…>
 <behavior>
<event>doubleClick(self,Mouse1)</event>
<action>
<transformationSystem …>
<lhs>
<window … mapId = "M1" name="registerWindow"…>
<textComponent … mapId ="M2" isEditable=true/></window>
</lhs>
<rhs>
<window … mapId = "M1" name="registerWindow"…>
<textComponent…mapId="M2" isEditable=true content=""/></window>
</rhs>
</transformationSystem> </action> </behavior>
…
</button>
```

Figure 10. Event Language in USIXML at the CIO level: Clicking on button 1 erases all editable textComponents of registerWindow

Navigation is part of a dialog specification; consequently it is easily described with dialog elements exposed above. Nonetheless, from previous works [36], we consider navigation definition as a pattern-based activity. USIXML provides an ad hoc relationship to define navigation in a straightforward way: `graphicalContainerTransition`. This relationship type enables to specify an open/close, suspend/resume, minimize/maximize relationship among containers populating an application.

4.7.3 Transformation model

A transformation model has been introduced to represent the possible transformations as defined in the framework of Fig. 1 (i.e., abstraction, reification, translation). Like actions in the behavior specification, transformations are performed via graph transformation rules as introduced in [20]. A model transformation is performed by one or several transformation system. Each transformation system realizes an identifiable design goal (i.e., widget selection, layout, navigation definition,…) in the transformation process. Fig. 12 shows a simple transformation (a translation) consisting in one single rule aligning vertically all widgets of a container. This rule has been design with the graph grammar editor AGG (http://tfs.cs.tu-berlin.de/agg/). Its textual equivalent in USIXML is shown in Fig. 11.

```
…
<translation id= "TL1" name = "squeezeDisplay" description= "this translation
vertically aligns all widgets of a container" >
<sourceModel> cui<sourceModel>
<targetModel>cui<targetModel>
<transformationSystem id = "TR1" name = "Transfo1" … >
<transformationRule id = "rule1" name "squeeze1" >
      <lhs>
            <box  mapId ="M1">
            <graphicalIndividualComponent ruleSpecificId"gi1" mapId =M2>
            </graphicalIndividualComponent>
            </box>
      </lhs>
      <rhs>
            <box  mapId ="M1">
            <graphicalIndividualComponent ruleSpecificId"gi1" mapId =M2
            glueHoriz="left" >
            </graphicalIndividualComponent>
            </box>
      </rhs>
      <nac>
            <box  mapId ="M1">
            <graphicalIndividualComponent ruleSpecificId"gi1" mapId =M2
            glueHoriz="left" >
            </graphicalIndividualComponent>
            </box>
      </nac>
</transformationRule>
</transformationSystem>
</translation>
…
```

Figure 11. Translation expressed in USIXML.

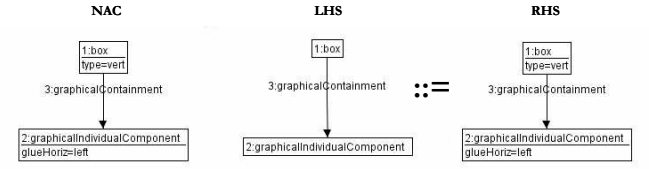


Figure 12. Aligning rule expressed in AGG in terms of a transformation.

5. CONCLUSION

In this paper, we have presented USIXML, a UIDL that addresses various requirements in UI design such as coverage of multiple models, relationships, and levels of abstraction. Graph transformations are explicitly used to define an executable mapping mechanism between these fragments so as to support continuous and seamless development of UIs from multiple entry points. USIXML is original and different with respect to existing UIDL regarding the following aspects:

- USIXML is precisely structured into four levels of abstraction that do not need all to be specified to obtain a UI.
- USIXML can be used to specify a platform-independent, a context-independent, and a modality-independent UI. For instance, a UI that is defined at the AUI level is assumed to be independent of any modality and platform. Therefore, it can be reified into different situations. Conversely, a UI that is defined at the CUI level can be abstracted into the AUI level so as to be transformed for another context of use.
- USIXML allows the simultaneous specification of multiple facets for each AIO, independently of any modality.
- USIXML encompasses a detailed model for specifying the dynamic aspects of UI based on productions (right-hand side, left-hand side, and negative conditions) and graph transformations. These aspects are considered as the basic blocks of a dialog model that is directly attached to the CIOs of interest, thus facilitating the local specification.
- Thanks to these dynamic aspects, virtually any type of adaptation can be explicitly specified. In particular, a transformation model consisting of a series of adaptation rules can be specified equally in an integrated way with the rest of the UI.
- USIXML contains a simplified abstraction for navigation based on windows transitions, that is compatible with dynamics.
- USIXML is based on Allen relationships for specifying constraints in time and space at the AUI level, that can be in turn mapped onto more precise relationships at the CUI level. These relationships are applicable to graphical UIs, vocal UIs, multimodal UIs, and virtual reality UIs.
- Similarly, a progressively more precise specification of the CIO layout can be introduced locally to concretize the Allen constraints imposed at the AUI level.
- USIXML defines a wide range of CIOs in different modalities of use so as not to be limited only to graphical CIOs.
- USIXML already introduced a catalogue of predefined, canonical inter-model mapping that can be expanded and a taxonomy of task types that facilitate the identification and selection of concepts at both the AUI and CUI levels.

From these advances, we can conclude that USIXML is probably one of the mostly integrated UIDL that addresses platform-, modality-, and context-independence and sensitivity. Depending on the kind of UI that is envisioned, USIXML can be used to specify only those parts that are required for a specific case.

ACKNOWLEDGEMENTS

The authors would like to thank Cameleon partners who contributed USIXML V1.2: Lionel Balme, Gaëlle Calvary, Cristina Chesta, Alexandre Demeure, Joëlle Coutaz, Jean-Thierry Lechein, Fabio Paternò, Stéphane Raymond, Carmen Santoro, and Youri Vanden Berghe. This paper presents USIXML V1.4, an extension of USIXML V1.2 with dialog model, more inter-model mappings, a context model made up of user, platform, and environment, and the concrete user interface level. Laurent Bouillon is supported by the "Cameleon" research project (http://giove.cnuce.cnr.it/ cameleon.html) under the umbrella of the European Fifth Framework Programme (FP5-IST2). Murielle Florins is supported by "Salamandre" research project (http://www.isys.ucl.ac.be/research/salamandre.html) under convention n°001/4511 of "Initiatives II" research program, Walloon Region (Belgium). Benjamin Michotte is supported by the SIMILAR network of excellence (http://www.similar.cc), the European research task force creating human-machine interfaces similar to human-human communication of the European Sixth Framework Programme (FP6-2002-IST1-507609). Daniela Trevisan is supported by the Mercator project.

REFERENCES

1. Ali, M.F., Pérez-Quiñones M.A., Abrams M., *Building Multi-Platform User Interfaces With UIML*, in A. Seffah & H. Javahery (eds.) Multiple User Interfaces: Engineering and Application Framework, John Wiley and Sons, 2003.
2. Allen, J.F., *Maintaining Knowledge about Temporal Intervals*, Communications of the ACM, Vol. 26, No. 11, November 1983, pp. 832-843.
3. Bouillon, L., Vanderdonckt, J., Chow, K.C., *Flexible Reengineering of Web Sites*, Proc. of 8th ACM Int. Conf. on Intelligent User Interfaces IUI'2004 (Funchal, 13-16 January 2004), ACM Press, New York, 2004, pp. 132-139.
4. Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., Vanderdonckt, J., *A Unifying Reference Framework for Multi-Target User Interfaces*, Interacting with Computers, Vol. 15, No. 3, June 2003, pp. 289-308.
5. Chamberlain, D., Angel Diaz, Dan Gisolfi, Ravi Konuru, John Lucassen, Julie Macnaught, Stephane Maes, Roland Merrick, David Mundel, TV Raman, Shankar Ramaswamy, Thomas Schaeck, Rich Thompson, and Charles Wiecha, *WSXL: a web services language for integrating end-user experience*, in Proc. of 3rd Conf. on Computer-Aided Design of User Interfaces CADUI'2002, Kluwer Ac., Dordrecht, 2002, pp. 35-50.
6. Chikofsky, E.J. and Cross, J.H., *Reverse Engineering and Design Recovery: A Taxonomy*, IEEE Software, Vol. 1, No. 7, January 1990, pp. 13-17.
7. Constantine, L., *Canonical Abstract Prototypes for Abstract Visual and Interaction Design*, in Proc. of 10th Int. Workshop on Design, Specification, and Verification of Interactive Systems DSVIS'2003, LNCS, Springer-Verlag, 2003.
8. Eisenstein, J., Vanderdonckt, J., Puerta, A., *Model-Based User-Interface Development Techniques for Mobile Computing*, Proc. of 5th ACM Int. Conf. on Intelligent User Interfaces IUI'2001 (Santa Fe, 14-17 January 2001), Lester, J. (Ed.), ACM Press, New York, 2001, pp. 69-76.
9. Elting, Ch., Zwickel, J.and Malaka, R., *Device-Dependent Modality Selection for User Interfaces – An Empirical Study*, in Proceedings of 6th Int. Conf. on Intelligent User Interfaces IUI'2002 (January 13-16, 2002, San Francisco), ACM Press,

New York.
10. Florins, M., Vanderdonckt, J., *Graceful Degradation of User Interfaces as a Design Method for Multiplatform Systems*, in Proc. of 8th ACM Int. Conf. on Intelligent User Interfaces IUI'2004 (Funchal, 13-16 January 2004), ACM Press, New York, 2004, pp. 140-147.
11. Gaeremynck, Y., Bergman, L.D., Lau, T., *MORE for Less: Model Recovery from Visual Interfaces for Multi-Device Application Design*, in Proc. of ACM Int. Conf. on Intelligent User Interfaces IUI'2003 (Miami, January 12-15, 2003), ACM Press, New York, pp. 69-76.
12. Larman, C., *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, Prentice Hall, July 2001.
13. Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, B., *TOMATOXML, a General Purpose XML Compliant User Interface Description Language*, TomatoXML V1.2.0, Working Paper n°105, IAG, Louvain-la-Neuve, 19 February 2004.
14. Limbourg, Q., Vanderdonckt, J., *Transformational Development of User Interfaces with Graph Transformations*, Proc. of 5th Int. Conf. on Computer-Aided Design of User Interfaces CADUI'2004 (Madeira, 14-16 January 2004), Kluwer Academics Pub., Dordrecht, 2004.
15. Luyten, K., Van Laerhoven, T., Coninx, K., Van Reeth, F., *Runtime Transformations for Modal Independent User Interface Migration*, Interacting with Computers, Vol. 15, No. 3, 2003, pp. 329-347.
16. Nichols, J., Myers, B.A., Higgins, M., Hughes, J., Harris, T.K., Rosenfeld, R., Pignol, M., *Generating Remote Control Interfaces for Complex Appliances*, Proc. of the 15th Annual ACM Symposium on User Interface Software and Technology UIST'2002, ACM Press, New York, 2002.
17. Paternò, F., *Model-Based Design and Evaluation of Interactive Applications*, Springer-Verlag, Berlin, 2000.
18. Paternò, F., Santoro, C., *One Model, Many Interfaces*, in Proc. of 3rd Int. Conf. on Computer-Aided Design of User Interfaces CADUI'2002, Kluwer Acad., Dordrecht, 2002, pp. 143-154.
19. Puerta, A. and Eisenstein, J., *Developing a Multiple User Interface Representation Framework for Industry*, in: A. Seffah & H. Javahery (eds.) Multiple User Interfaces: Engineering and Application Framework, John Wiley and Sons, 2003.
20. Souchon, N., Vanderdonckt, J., *A Review of XML-Compliant User Interface Description Languages*, Proc. of 10th Int. Conf. on Design, Specification, and Verification of Interactive Systems DSV-IS'2003, Lecture Notes in Computer Science, Vol. 2844, Springer-Verlag, Berlin, 2003, pp. 377-391.
21. Trevisan, D., Vanderdonckt, J., Macq, B., *Analyzing Interaction in Augmented Reality Systems*, Proc. of ACM Multimedia 2002 International Workshop on Immersive Telepresence ITP'2002 (Juan Les Pins, 6 December 2002), Pingali, G., Jain, R. (Eds.), ACM Press, New York, 2002, pp. 56-59.
22. Vanderdonckt, J., Limbourg, Q., Florins, M., *Deriving the Navigational Structure of a User Interface*, Proc. of 9th IFIP Conf. on Human-Computer Interaction INTERACT'2003 (Zurich, 1-5 September 2003), M. Rauterberg, M. Menozzi, J. Wesson (Eds.), IOS Press, Amsterdam, 2003, pp. 455-462.
23. 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, 24-29 April 1993), ACM Press, New York, 1993, pp. 424-429.
24. Vanderdonckt, J., Berquin, P., *Towards a very large model-based approach for user interface Development*, in Proc. of 1st IEEE Int. Workshop on User Interfaces to Data Intensive Systems UIDIS'99, IEEE Computer Society Press, Los Alamitos, 1999, pp. 76-85.
25. Wong C., Chu H.H. and Katagiri M.A., *Single-Authoring Technique for Building Device-Independent Presentations*, in Proceedings of W3C Workshop on Device Independent Authoring Techniques (St. Leon-Rot, 15-26 September 2002).

Incorporating UIDLs into Model-Driven Development

Xiaoping Jia, Adam Steele
DePaul University, School of CTI
243 S. Wabash Avenue,
Chicago, IL 60604
[xjia,asteele]@cs.depaul.edu

ABSTRACT
In Model-Driven Development (MDD) the focus in on defining an abstract model of the program rather than the program's code. MDD has the potential to deliver greater cost savings to software developers by automating much of the most time-consuming and error-prone aspects of software development including code-generation and testing. However, there are obstacles to realizing the full benefits of MDD. The most critical of these is that current modeling notations are incomplete in that they do not support important aspects of the program such as the user interface (UI) and extra functional requirements (XFRs). We propose set of formal notations to describe a loosely coupled model of three aspects of an application (structural, behavioral, and UI) and an event-based framework to tie the model together. We will also develop knowledge-based tools to automate the compilation of the model. The key to our approach is capturing a human expert's knowledge of software architecture and design principles, so that we can codify and represent this knowledge in automated model compilation tools.

Keywords
Model-Driven Development, UIDL, Model Compilation, Knowledge-based Software Engineering

INTRODUCTION

The defining characteristic of model-driven development (MDD) is the building of models rather than of programs [19, 12, 23]. MDD shifts the development focus from programming language code to models, such as those in UML-2 [21]. MDD has the potential to deliver greater cost savings to software developers by automating much of the most time-consuming and error-prone aspects of software development including code-generation and testing. These savings are compounded when we consider that we might have a single application that we wish to be executed on many diverse platforms, each with their own constraints and display challenges. Similarly, MDD supports other major goals of software developers such as fast turnaround times and higher quality. By encapsulating the code generation process within an easy-to-change model, developers can quickly modify applications to include new functionality. Because of MDD's potential benefits, companies are already working to deliver supporting technologies[20].

We define a *model* to be "a consistent and complete set of formal elements describing a system that is amenable to analysis" which is consistent with definitions given in [16] and [22]. The current modeling language of choice is UML-2, which allows software developers to design software systems from a number of different yet related views: the user view (functional requirements), structural view (static constructs such as classes and relationships), behavioral view (interactions between the structural elements), configuration view (components or subsystems), and environment view (deployment and installation). Each view represents a set of concerns within the overall software system. When combined, these views attempt to provide a 360◦ view of the software system.

While UML-2's views are a good first step, we do not believe them to be sufficient for true MDD. We argue that the true benefits of MDD require models that are abstract, understandable, formal, analyzable and complete.

Abstract Abstraction is a key to managing the complexity inherent in even the simplest applications. One desirable property of software models is to be able to express the models at a high level of abstraction and to be able navigate between different levels of abstraction, or *views*.

Understandable The models used to support MDD must facilitate the understanding and communication of the application domain without introducing unnecessary complexity.

Formal For models to be analyzed and evaluated and to generate complete applications from the models, it is necessary that all of the requirements are precisely and formally articulated, formulated and verified.

Analyzable Formal models are amenable to a variety of mathematical analysis and manipulation, including animation and formal verification.

Complete To generation complete applications from models, the models must be complete. For a model to be *complete*, it must describe three sets of requirements: functional requirements, extra-functional requirements (XFRs)[1] and user interface (UI) requirements. This proposal describes the ZOOM approach, which consists of a set of notations and supporting tools that attempts to

[1] These are also called *non-functional requirements*

provide an infrastructure that delivers all of these capabilities.

Model Transformation and Compilation

Models at a high level of abstraction focus more heavily on application domain concepts and are thus less sensitive to changes within the technical domain, which is often the model representation used at lower levels of abstraction. Thus *model transformation*, the process of converting a model from one level of abstraction to another, is of critical importance, particularly when we consider that not only should a model be able to capture and appropriately transform functional requirements, but also the extra-functional and environmental requirements that act as systemic constraints. In the case of MDD, the realization of an application from a set of models requires the transformation of a platform-independent model (PIM) into a platform-specific model (PSM) usually via some transformation language. According to Weis, *et. al.* [35], such languages are either imperative or rule-based. Imperative transformations require some kind of language, whereas rule-based languages can be expressed using a visual notation, thereby facilitating the understandability of the transformation process. Sendall, *et. al.* [24], identify three approaches for defining imperative transformations: direct model manipulation, intermediate representation, and transformation language support. A more compelling approach is the use of *model compilation* where the model is transformed into a completed application, not through the use of transformation languages, but through the application of rules codified within a software engineering knowledge-base, i.e., a rule-based approach to transformation. Such an approach removes the "programming" aspect from the transformation; model transformations can be changed by updating the rules within the knowledge-base.

OBSTACLES

There are many obstacles to realize the full benefits of model-driven development. The most critical obstacles are inadequate modeling notations and tool support.

Inadequate Modeling Notations

UML-2 is the *de facto* standard object modeling notation for software engineering. It allows modelers to capture a wealth of information about software system components, their behaviors, and their interactions. However, UML-2 is insufficient for realizing the full benefits of MDD owing to several critical deficiencies:

Incomplete UML-2 is an incomplete notation. Many UML-2 tool vendors have to provide a means for modelers to "inject" actual implementation language details into the models in order to describe significant detail. In addition, UML-2 does not address XFRs although work has been done to include some XFR elements into the notation [7, 8]. Similarly, the OMG has published a standard [3, 4] for UML-2 profiles that specify schedulability, performance, and timing requirements in real-time systems. Another OMG RFP and its proposals [2, 10] address quality of service constraints in general. A significant omission in UML-2 is that it does not specify anything with respect to the User Interface. Work has been done by the W3C to design a draft Multimodal Interaction Framework [30], and specify a multimodal markup language, EMMA [28] but the efforts are preliminary and not supported by a significant executional framework.

Semi-Formal UML-2 is not completely formal. UML-2 is based on the Meta-Object Facility (MOF) [13] defined by the OMG. MOF defines the structural constraints formally, but it does not define its semantics formally. More importantly, although MOF does define an Object-Constraint Language (OCL) [14], its use in UML-2 is optional and tends to be treated as documentary rather than as a set of imperatives

Inconsistent UML-2 only provides limited mechanisms to ensure that the various views are consistent with one another or indeed that the elements are consistent within one view. Several UML-2 diagrams can show almost identical perspectives of the same semantics, such as the collaboration and sequence diagrams. Such redundancy can result in different views of the same model becoming inconsistent. It is currently up to the various tool vendors to ensure that such inconsistencies are not introduced.

Inadequate Tool Support

The OMG is focused on using standards such as XMI [27, 26] and XSLT [33] to provide support for the transformation from their platform-independent models (PIM) to the platform-specific models (PSM). The idea of transformations is not new and closely mirrors the approach used by code generators in the past. Such transformation techniques have the benefit of delivering significant amounts of code quickly, while at the same time requiring very little detailed information about the target architecture on the part of the modeler. However, the existing OMG transformation approaches only address some aspects of the finished application. For example, the extra-functional requirements are as yet unaddressed. Considering their importance to the overall application, this is a significant problem.

In addition, the existing transformation process focuses heavily on XSLT-based technologies. Designing transformations using XSLT is a detailed and tedious process. In addition, it is difficult to combine different transformation rules or to synthesize new rules based on the existing ones. All such capabilities must be encoded into the transformation in order for them to be used during the transformation process. This can result in a significant amount of logic being embedded within the transform that might easily rival the amount of logic needed to build the application itself.

Even though there are many industry-standard tools that support UML-2, to our knowledge, few, if any, of these tools support formalism. Often, OCL is either not required at all, or is only used to provide supporting documentation. It is rarely if ever used during the code generation process. Support for model compilation is *ad hoc* and relies heavily on UML-2 profiles or templates. These are often outside the modeler's direct control making it difficult to change the model transformation process. Another area we feel to

be weak in existing tools is the support for action semantics. In most cases, modeler actions must be specified in some procedural language. This leads to a "mix-and-match" approach which makes the model more difficult to understand and maintain as well as being less resilient to change.

PROPOSED APPROACH

Our vision for MDD centers on a) approaches and notations for capturing formal and complete models of all aspects of applications, and b) knowledge-based approaches and tools that convert models into complete applications through a process known as *model compilation*.

Formal and Complete Models

To support fully automated model compilation, verification and validation, it is essential that each model be formally specified. By ensuring that all models be defined in a formal way, it is possible to statically check those models before any code is written or generated. This analysis can tell software developers whether or not the model meets all of the documented requirements. Errors, omissions, and contradictions in the model can also be located before any significant time has been spent in constructing the solution resulting in a savings of time, money, and developer resources. An *animator* can hypothesize or predict the behavior of method bodies based on the formal pre- and post-conditions defined for the model. A virtual "execution" of the model is then performed to help the designers identify and correct any deficiencies. Through animation, modelers can gain critical insights into the behavior of the application before it has been completed. Such insights might focus on otherwise "invisible" properties of the application such as concurrency or recoverability. By identifying problems earlier in the development cycle, additional functional, extra-functional, or environmental requirements can be specified and added to the model to correct them.

Knowledge-Based Supporting Tools

Model compilation, the process of generating a completed application that exposes the functional, extra-functional, and environmental requirements described by its model, is a critical element of MDD. However, this compilation process is more complex than mere language compilation. Not only must the generated code meet all of the functional requirements identified by its model, but the compilation process must also address issues such as the choice of architecture, data structures, and algorithms. To support such decisions, we believe that the model compilation process must be knowledgebased. The key to our approach is attempting to capture the software architecture and design knowledge of human experts, codifying and representing these knowledge, and applying these knowledge in knowledgebased tools to carry out fully automated model compilation. We believe that this approach offers the greatest advantages in terms of being able to a) build up and maintain the knowledge-base gradually and incrementally, b) resolve often competing goals, especially in the areas of XFRs and UIs, and c) reach a solution that will satisfy all of the final application's requirements across all three aspects (structural, behavioral, and UI).

MODELING WITH ZOOM NOTATION

Overview

To overcome the obstacles associated with the existing modeling notations and to realize our vision of MDD, we have developed a new formal modeling notation is called Z-based Object-Oriented Modeling notation or *ZOOM*, which is based on the formal specification notation Z [25, 37, 36], and several key components of UML-2. Although widely used to specify software systems, one of the deficiencies of Z is that it does not provide useful mechanisms to support object-oriented modeling such as classes or inheritance. ZOOM brings formal foundations to object-oriented notations by providing textual and graphical representations of models that are consistent with UML-2 and that can be checked formally for consistency. It allows constraints such as preconditions, postconditions and invariants to be specified formally. To make it easy for practitioners to use,

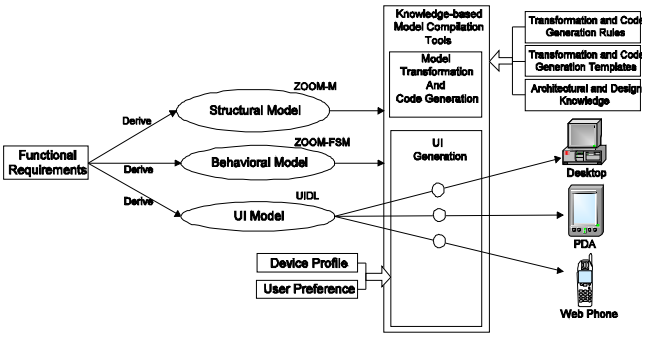


Figure1: An overview of the ZOOM approach

ZOOM adopts a syntax that is similar to commonly used programming languages such as Java and C++. Unlike UML-2, which separates the system into several loosely related views, ZOOM provides a mechanism to integrate those views together along with their formal semantics. This approach overcomes some of the problems with intra- and inter-model consistency within UML-2[11]. ZOOM separates an application into three parts: structure, behavior and user interface. ZOOM provides three separate but related notations to describe each of these three key aspects: ZOOM-M for structural models, ZOOM-FSM for behavioral models, and a User Interface Description Language (ZOOM-UIDL) for user interface models. An event-based framework integrates the different parts of each ZOOM model. The main reasons for adopting three different notations are a) these three aspects have distinct characteristics; and b) they are often changeable independent from the others. Therefore it is beneficial to separate these three aspects, make them loosely coupled, and model each aspect using a notation that is most suitable for that aspect. Figure 1 shows the structure of a ZOOM model. The functional requirements derive the structural and behavioral models. The user interface design is derived from the functional and UI specific

requirements. UI requirements often include the "look and feel", user profile and display device limitations. These UI requirements are expressed as *user preferences* and *device profile*, which are used by the knowledge-based model compilation tools (see below) to generate different UI implementations. The separation of a system into loosely coupled structural, behavioral and UI models is a well-known maxim in software engineering, "separation of concerns", which separates the system based on special purpose concerns[6]. The three aspects are often changeable independent from the others. This separation allows each aspect of the system to be specified separately, making each aspect easier to specify, understand, and change while reducing the coupling with the other aspects. Under this separation, modelers can easily modify the user interface based on the profile and user preferences without changing the structural or behavioral models. Another advantage of this separation is that we use different formal notations to describe the different aspects of the system. Because the structural, behavioral, and UI models describe different parts of system, they have their own distinct characteristics. It is beneficial to use a notation designed specifically for the needs of each model. Significant work has already been complete on the ZOOM notations. The formal syntax and semantics along with a set of basic support tools are available.

Structural Model

Structural models in ZOOM are object-oriented models with classes, relations, and formal specifications of the functionality of the entities. The notation for the structural model, ZOOM-L, is a Z and OCL based language with a Java/C++ like syntax to make it easy for practitioners to adopt. ZOOM-M is formal, object-oriented and side-effect free. ZOOM-M is strongly typed with a semantically rich type system that supports inheritance and generic types [15]. ZOOM-M includes a library of pre-defined structural models including: Set, List, OrderedSet, Bag, Relation, Map, Pair and Tuple. ZOOM-M is as expressive as Z and OCL with a friendlier syntax and a much richer set of libraries compared to Z and OCL. Each ZOOM model has dual textual and graphical representations. The graphical views of ZOOM structural models are consistent with UML-2 package and class diagrams along with the formal specifications including operation preconditions and postconditions as well as class invariants [18, 17, 34]. As an example, consider an instant-messenger system. Some of the entities might include: User, FriendList, Contact, Log, PersonalInformation and Messenger. As with other object-oriented design technologies, these entities should be specified as autonomous, precisely specified *structs*, which are class-like entities and the basic building blocks in ZOOM-M, with proper attributes and operations. Part (a) in Figure 2 shows the graphical view of a simplified structural model for the above scenario. A set of supporting tools has been built for ZOOM, including, parser, type checker, interpreter, animator, and an automated theorem prover (ATP). The interpreter evaluates and executes the ZOOM-M expressions and scripts. The animator virtually "executes" the structural models by introducing a default model implementation that is consistent with the formal specification. The ATP is optimized for ZOOM-M and conducts the proof obligations defined in ZOOM-M specifications. It provides a powerful reasoning mechanism for further static analysis.

Behavioral Models

The behavioral model is the central communication mechanism that links the structural and UI models. It uses a formalized state diagram to specify the dynamic aspects of a system. The textual specifications are described by ZOOM-FSM. Graphically, the behavioral model is consistent with a UML-2 state chart [1]. ZOOM-FSM

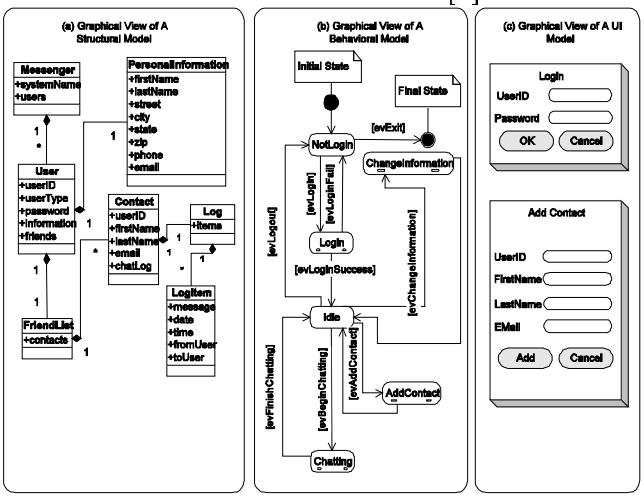


Figure 2: An example of ZOOM Models

features a rich syntax, a mechanism to use the structural models specified in ZOOM-M, and a strong compatibility with UML-2 state chart including composite/concurrent states, state history and state synchronization. ZOOM-FSM shares the same type and expression system as ZOOM-M. In the instant-messenger example, the behavioral model should include several finite state machines for different user behaviors. Part (b) in Figure 2 shows part of a simplified behavioral model. An experimental ZOOM-FSM translation engine for code generation and animation of the behavioral models has been developed. The translation engine takes the behavioral models and the related structural model and generates executable Java code. The translation engine also provides an underlying architecture for this executable code with built in support for persistence, signals and event generation, transition and state priorities, threading and concurrency.

User Interface Model

Unlike UML-2, ZOOM separates UI models from other aspects of the program. In its present form, UML-2 fails to enforce this separation of concerns; such enforcement is up to the modeler. While, the UI typically has a tight association to a specific platform, the structural and behavioral aspects are typically platform independent [9].

This means it is important to decouple the UI design from the other aspects of the application. ZOOM currently uses a specific UI modeling notation to achieve this goal. This decoupling of the specification of the UI's characteristics from its implementation allows us to target different platforms (e.g. desktop, handheld, speech) quickly and easily. In keeping with the ZOOM philosophy, we use rules to capture the expert knowledge of the designer in order to optimally present the UI on a particular device. In order to specify the design of a UI in a formal way, we use a User Interface Description Language, ZOOM-UIDL. ZOOM-UIDL is an XML[29, 5] based language whose structure is defined by a set of schemas according to *XML Schema* [31, 32]. ZOOM-UIDL allows UI designer to specify fundamental interactions between the user and the interface and the attributes and events related to each UI component that will achieve this interaction. An example of this type of approach is shown in Figure 1. In the current implementation of ZOOM, the interaction model is defined by ZOOM-UIDL; each UI component is specified according to a schema that defines ZOOM-UIDL. Each schema also binds to a struct definition in the structural model. The binding of the entire set of schemas of ZOOM-UIDL results in a set of struct definitions known as the *User Interface Structural Model (UISM)*. For example, a button component in a user interface is specified according to the `Button` schema in ZOOM-UIDL. The `Button` schema binds to `UIButton` struct in the structural models. As an example of this we have specified the interface behavior of an instant messaging system. In this example the user interface includes **Login**, **AddContact**, **Chatting**, **ChangePersonalInformation** dialogs. The most critical one is the **Chatting** interface. Part (c) in Figure 2 shows part of a simplified UI model in its graphical view, including the **Login** and **AddContact** dialogs. To demonstrate the feasibility and advantage of our UI modeling approach, a rule-based tool has been developed, which successfully implements two separate sets of rules for Java Swing and SWT frameworks. Part of our current research effort is to generalize the interaction model to encompass multiple modalities (consistent with the W3C Multimodal Interaction Framework) and to develop the transformation rules in order to target displays with different form factors (e.g. desktops, handhelds, etc.), and in particular Speech User Interfaces (SUI) defined by VoiceXML.

We provide an example below in Figures 3 and 4 that shows our early approach to modeling an interface that is implemented in both the Swing and SWT frameworks.

```
<?xml version="1.0" encoding="UTF-8"?>
<Start name="SplitPane" >
    <Window title="SplitPane" name="SplitPane"
            visible="true" show="true">
        <SplitPane right="right" left="left">
            <Panel name="left">
                <BoxLayout axis="1"/>
                <List>
                    <ListItem element="apple"/>
                    <ListItem element="orange"/>
                    <ListItem element="pineapple"/>
                    <ListItem element="grape"/>
                    <ListItem element="peaches"/>
                    <ListItem element="watermelon"/>
                    <ListItem element="pear"/>
                    <ListItem element="plum"/>
                </List>
                <ComboBox event="Action">
                    <Item element="apple"/>
                    <Item element="orange"/>
                    <Item element="pineapple"/>
                    <Item element="grape"/>
                    <Item element="peaches"/>
                    <Item element="watermelon"/>
                    <Item element="pear"/>
                    <Item element="plum"/>
                </ComboBox>
            </Panel>
            <Panel name="right">
                <FlowLayout/>
                <Image imagesrc="sample.jpg"/>
            </Panel>
        </SplitPane>
    </Window>
</Start>
```

Figure 3: ZOOM-UIDL Definition

Swing Split Pane

SWT Split Pane

Figure 4: Interfaces generated from ZOOM-UIDL in Figure 3.

An Event-Based Framework for Model Integration

We propose an event-based integration framework to integrate and coordinate the activities among the structural, behavioral and UI models. Figure 5 shows the structure of event-based framework. The shaded part indicates the run-time components of the framework and the unshaded elements indicate the design-time components. At run-time, each UI component is automatically bound to an instance in the structural model. When a user triggers a UI event, this event is transmitted to the appropriate behavioral model instance, which may trigger some state transitions. The behavioral model instances can then manipulate the structural model instances and changes to instances bound to UI components will cause UI components being updated automatically. Each component only directly interacts with one other component, as indicated in Figure 5. This event-based framework integrates the three models while maintaining simplicity and loose coupling and without limiting the capabilities of the applications.

KNOWLEDGE-BASED MODEL COMPILATION

We realize that a key to successful code generation and model compilation is that it requires a tremendous amount of knowledge in software architecture and design. A critical component of our approach is to investigate and develop tools for knowledge-based model compilation. The amount of knowledge that goes into even the most basic architectural decision is staggering and encompasses many

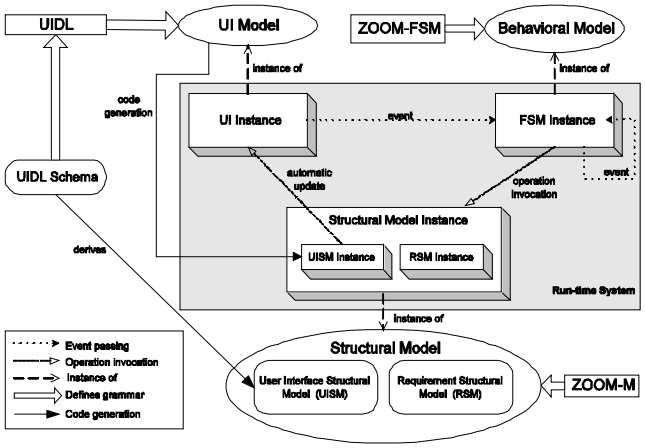


Figure 5: An event-based framework for model integration

different areas including: available tools or frameworks, time and other resource constraints, target platform, perceived risks such as future enhancements, familiarity with trade-offs such as time-space and awareness of technical pitfalls. These issues can affect every decision during a software development effort from its inception to its delivery. The acquisition and representation of this knowledge is a complex process, particularly since there is no agreed upon solution or even criteria for any of these decisions. Each software architect will bring their own "baggage" to any software development effort that will often bias their decisions or at least their thought process. In order to make our approach a reality, and to make it more than just an improved code generation utility, we must address three critical problems: knowledge acquisition, knowledge representation, and knowledge utilization. Software architecture and design knowledge is acquired from human experts. The acquired knowledge will be codified and stored in a knowledge-base. The purpose of this knowledge-base is to guide the automated decision making process to derive and/or select a set of transformation rules and templates that are used to transform the models into complete applications. Conceptually, our vision for ZOOM is shown in Figure 6. Models, parameters and environmental requirements are provided to the model compilation subsystem. This subsystem produces various model transformations by tapping into a software architecture and design knowledge-base. The completed model transformations may yield several candidate implementations. These implementations are then evaluated, either through a human or automated agent, and a final application implementation is selected. This implementation will satisfy all of the application's functional and environmental requirements, and will satisfy the extra-functional requirements within some specified tolerance. The first key problem in constructing the knowledge-base is the acquisition of the knowledge in the first place. Not only must information be collected about appropriate mechanisms to transform models into completed implementations, but also in the qualities of those implementations from an architectural and environmental perspective. We expect that this knowledge will stem from multiple sources including analytical methods as well as best practices in software architecture and UI design. The second critical problem is determining how the knowledge gathered during the acquisition phase is to be represented within the knowledge-base. This is important since our approach will draw heavily on this knowledge and thus requires it to be formulated in such a way as to be easily accessible as well as to adequately facilitate reasoning and decision making during the model transformation phase. When a model is transformed into a working implementation, the knowledge that has been gathered into the knowledge-base must be utilized through a formal reasoning and decision making process that result in a set optimal transformations and templates being synthesized or selected. These transformations and templates will be applied to the model to generate the a complete implementation that satisfies not just its functional requirements, but also the extra-functional and environmental requirements as well.

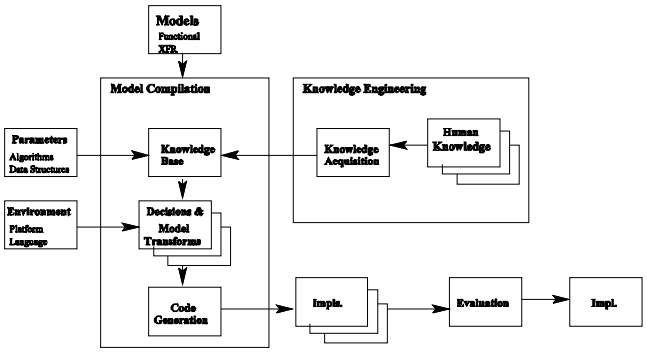


Figure 6: The architecture of knowledge-based model compilation

RESEARCH PROBLEMS

A substantial amount of existing work has resulted in a set of basic supporting tools for our approach; and we are currently focused on investigating and researching solutions to the following problems.

- Notations, approaches, and processes for formal and complete models of software applications. Refine and extend the ZOOM approach to address challenges in modeling the user interface, and extrafunctional requirements. Investigate approaches to model user interface and extra functional requirements at an appropriate level of abstraction and in an implementation independent fashion.
- A knowledge-base of software architecture and design knowledge to facilitate automated model compilation. Acquire and codify the necessary knowledge in software architecture and design. Investigate the appropriate representation of the knowledge that adequately facilitates easy access, reasoning, and decision making. Investigate appropriate mechanisms for reasoning and decision making in the software architecture and design knowledge-base.
- A knowledge-based approach and mechanism that utilizes the software architecture and design knowledge-base to carry out automated model compilation.
- A generalized interaction model that encompasses multiple modalities (consistent with the W3C Multimodal Interaction Framework) and transformation rules that target displays with different form factors (e.g. desktops, handhelds, etc.), and in particular Speech User Interfaces (SUI) defined by VoiceXML.

CONCLUSION

We have proposed an extension to the existing approaches to Model-Driven Development that incorporates an XML based, independent representation of the User Interface and Extra-Functional Requirements. These descriptions provide a more complete picture of the application that ultimately can be transformed in to an executable form by the use of knowledge-based tools.

ACKNOWLEDGMENTS

We thank the students who have participated in the SE and HCI seminars at DePaul University for their input and inspiration.

REFERENCES

1. UML™ 2.0 Superstructure Specification. OMG Document ptc/03-08-02 (August, 2003).
2. UML™ Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms. OMG Document ad/02-01-07 (January, 2002).
3. UML™ Profile for Schedulability, Performance, and Time RFP. OMGDocument ad/99-03-13 (March, 2003).
4. UML™ Profile for Schedulability, Performance, and Time Specification. OMG Document ptc/02-11-01 (November, 2002).
5. Neil Bradley. *The XML Companion, 3rd Ed.* AddisonWesley, 2001.
6. W. Hrsh C. Lopes. Separation of concerns. Technical report, Northeastern University, 1995. Boston, MA.
7. Luiz Marcio Cysneiros and Julio C¨esar Sampaio do Prado Leite. Using UML to reflect non-functional requirements. In *Proceedings of the 2001 conference of the Centre for Advanced Studies on Collaborative Research*, page 2. IBM Press, 2001.
8. Luiz Marcio Cysneiros and Julio C¨esar Sampaio do Prado Leite. Non-functional requirements: from elicitation to modelling languages. In *Proceedings of the 24th international conference on Software engineering*, pages 699–700. ACM Press, 2002.
9. Paulo Pinheiro da Silva and Norman W. Paton. User interface modeling in umli. *IEEE SOFTWARE*, 20(4): 62–69, 2003.
10. Miguel A. de Miguel. Qos modeling language for high quality systems. In *Proceedings of The Eighth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003)*, pages 210–216. IEEE, 2003.
11. William E.McUmber and Betty H. C. Cheng. A general framework for formalizing UML with formal languages. In *Proceedings of the 23rd International Conference on Software engineering*, pages 433–442, Toronto, Ontario, Canada, November 2000.
12. David S. Frankel. *Model-Driven Architecture: Applying MDA to Enterprise Computing*. JohnWiley & Sons, New York, NY, 2003.
13. Object Management Group. Meta-Object Facility 1.4. http://www.omg.org/technology/documents/formal/mof.htm.

14.Object Management Group. UML 2.0 OCL, 2003. http://www.omg.org/docs/ad/03-01-07.pdf.

15.X. Jia. The zoom notation - a reference manual. Technical report, DePaul University, 2004. Chicago,IL.

16.Stephen J. Mellor, Anthony N. Clark, and Takao Futagami. Model-driven development. *IEEE Software*, 20(5):14–18, 2003.

17.Bertrand Meyer. Applying 'design by contract'. 25(10): 40–51, oct 1992.

18.Bertrand Meyer. *Object-Oriented Software Construction, 2nd Ed.* Prentice Hall PTR, Upper Saddle River, NJ, 1997.

19.J. Mukerji and J. Miller. Model-Driven Architecture. http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01.1

20.J. Poole. Model-driven architecture: Vision, standards, and emerging technologies. In *ECOOP'01–Object-Oriented Programming*, 2001.

21.James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language reference manual*, 1998.

22.Ed Seidewitz. What models mean. *IEEE Software*, 20 (5):26–32, 2003.

23.Bran Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, 2003.

24.Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003.

25.J. M. Spivey. *The Z Notation: A Reference Manual, 2nd Ed.*, 1992.

26.Gary C. Doney Timothy J. Grose and Stephen A. Brodsky. *Mastering XMI: Java Programming with XMI, XML, and UML*. John Wiley & Sons, 2002.

27.DSTC Oracle Co. Platinum Technology Inc. Fujitsu Softeam Recerca Informatica Daimler-BenzUnisys Co., IBM Co. Xml metadata interchange (xmi) version 1.1. http://www.omg.org/docs/ad/99-10-02.pdf.

28.TheWorldWideWeb Consortium (W3C). Emma: Extensible multimodal annotation markup language. http://www.w3c.org/TR/emma/.

29.The World Wide Web Consortium (W3C). Extensible markup language (xml) 1.0 (third edition). http://www.w3.org/TR/2004/REC-xml-20040204/.

30.The World Wide Web Consortium (W3C). W3C multimodal interaction framework. http://www.w3.org/TR/mmi-framework/.

31.The World Wide Web Consortium (W3C). Xml schema part 1: Structures.http://www.w3.org/TR/xmlschema-1/.

32.The World Wide Web Consortium (W3C). Xml schema part 2: Datatypes. http://www.w3.org/TR/xmlschema-2/.

33.The World Wide Web Consortium (W3C). Xsl transformations (xslt) version 1.0.http://www.w3c.org/TR/xslt.

34.Jos Warmer and Anneke Kleppe. *The Object Constraint Language*. Addison Wesley, Boston, MA, 1999.

35.Torben Weis, Andreas Ulbich, and Kurt Geihs. Model metamorphosis. *IEEE Software*, 20(5):46–51,2003.

36.J. Woodcock and J. Davies. *Using Z Specification, Refinement, and Proof*. Prentice Hall Europe, 1996.

37.J. B. Wordsworth. *Software Development with Z*. Addison Wesley, Boston, MA, 1992.

The AMF Architecture in a Multiple User Interface Generation Process

Kinan Samaan, Franck Tarpin-Bernard
Laboratoire ICTT, Lyon
21, Av. Jean Capelle, 69621 Villeurbanne cedex - FRANCE
kinan@ictt.insa-lyon.fr, franck.tarpin-bernard@insa-lyon.fr

ABSTRACT
In the context of Multiple User Interface (MUI) generation, this paper presents the AMF architecture on which a method relies for the adaptation of interactive applications to the specific characteristics of a targeted context. In our model-based approach, we use a library of task patterns and interaction patterns to adapt the interaction model of the application.

For the description of AMF architecture, we use an XML file that ensures the link between the tasks model and the functional core of the application. An engine parses and processes the file to run the application.

Keywords
Multiple User Interface, design patterns, model-based, AMF, XML based language.

INTRODUCTION
Everyday, new platforms are emerging with new characteristics and new interaction capabilities. The traditional classification (PC, mobile phone, PDA, interactive TV…) is not sufficient to generate MUI for interactive applications. We must not propose the same interface for a PDA with a small coloured screen and a keyboard and another one with a larger screen and a stick.

These last years, many researches have been led on Multiple User Interface (MUI) generation processes. The model-based approaches seem to be the most promising. In classical software engineering, models like MVC [4] have been exploited for a long time. Recently, user models or task models have been introduced to help for the generation of MUI. Whereas these models can be easily described with XML, it is not the case with MVC. According to us, it is very important to be able to also model with a portable file the interaction model.

Indeed, the interaction model is one of the most important models to consider because, on the one hand, it manages the interaction between the user and the application, and on the other hand, it ensures the link with the functional core of the application. This model was often neglected in the steps of MUI generation.

In this paper, we present the basis of a new approach that integrates the interaction model and the platform model into the design and generation processes.

The adaptation process forces the designer to clarify/explain the links between the task model, the interface model and interaction model. For the description and the adaptation of the interaction model, our approach relies on the AMF architecture (Agent with Multiple Facets), which has been created in 1997 for modelling common interactive applications [12]. Indeed, AMF presents the following advantages that will be deeply discussed:

- The multi-facets concept is very interesting especially for multiple presentation definitions.
- The XML description of AMF models allows the definition of an abstract interaction model and patterns of interaction.
- A run-time engine is able to execute an AMF model and allows switching dynamically from a specific model or sub-model to another.

RELATED WORKS
To design and implement MUI [9] interactive software, the first approaches were based on description languages like UIML [1] and XIML (RedWhale) [15]. These languages organize adaptation processes in two levels or steps: an abstract level and a concrete level corresponding to the implementation in HTML, Java or WM. If this approach certainly represents a progress, we think that the proposed abstractions are still insufficiently generic. Mainly it imposes a particular style of interaction, i.e. this abstract level specifies, for example, a button that will be concretised under different forms (aspect, position, etc.), but mandatory imposes the use of button while neglecting the other forms of interaction more adapted for a given platform as the vocal recognition or use of physical button. In this way, abstract level is portability oriented and not plasticity [9] oriented.

Newer approaches try to define a component-based framework that will allow runtime migratable user interfaces, which are independent of the target software platform, the target device and the interaction modalities [5]. In these frameworks, the user interfaces are merely considered as a presentation of a single service or of more

functionally grouped services. These kinds of solutions are more powerful than the language-based ones but, as they do not use task models, they are not able to filter the functions that cannot be used in a specific context.

Other approaches mainly focus on the task model [7][11]. They filter a generic task model in order to define an abstract user interface and later build a concrete user interface. The Abstract user interface is described in terms of Abstract Interaction Objects (AIOs) [14] that are latter transformed into Concrete Interaction Objects (CIOs) once a specific target has been selected. Calvary et al. defined a unifying reference framework for multi-target user interfaces [2]. This framework tries to give a global view of the multiples approaches on MUI.

The improvement is important. However, these methods do not explicitly define an interaction model. As a consequence, they are very efficient for modelling basic interactions but are limited for modelling more sophisticated ones like "drag and drop". Currently they are dedicated to graphical interaction and need to be extended to manage multimodal and multi-style interactions.

THE GENERAL APPROACH OF MUI GENERATION

To allow a more important variation at the interaction style level as well as at the implementation level, it is necessary to introduce a richer and generic description and to replace the language-based approach by an architecture-based approach [10]. In the AMF approach, we propose to start with a task model and to map it to an architecture-based abstract interaction model expressed in AMF, then to concretise this one in relation to the characteristics of the working platform. Once the concrete interaction have been chosen, the degrees of freedom available allow an ultimate adaptation to the user and the environment.

Our approach consists in organizing the MUI generation process in 4 phases (Fig 1):

- Abstract application definition phase,
- Interaction styles selection phase,
- Concrete interface generation phase,
- Final adaptation phase.

The first phase consists in modelling the generic task model and the abstract interface model, and defines the links between these two models and the abstract interaction model of the application. The designer builds these models once for all.

The second phase aims at dynamically generating the components of the interface that are adapted to the target. This phase is activated when a target (that is a triple < user, environment, platform > [13]) is running the software. The process consists in transforming the previous models with an adaptation engine. For the adaptation, this engine considers two extra components: the platform model and a library of task and interaction patterns.

We can summarize the work of the adaptation engine in three points:

- It removes non-realizable tasks from the generic task model (e.g. removes a "print" task if the system does not detect a printer connected to the target). At this stage the engine also removes the elements of the abstract interface that are closely related to the removed tasks.
- According to the input devices of the target, the mechanism replaces each abstract task by a concrete task using a "task patterns" library (e.g. moving element with a pointing device).
- In parallel, the engine enriches the XML description of abstract interaction model by inserting the patterns that are associated to the task patterns using an "interaction patterns" library.

The third phase aims at generating a concrete interface where all the resources that will be used are selected but where the final parameters (layout, colour, volume…) are not set.

According to the characteristics of the devices (size, resolution, capacity…) and the user preferences, a second engine selects among potential resources for each element of the semi-concrete interface, the ones that are more appropriate to the circumstances of use. The dependencies that have been defined between the domain objects are considered so that the choices are coherent.

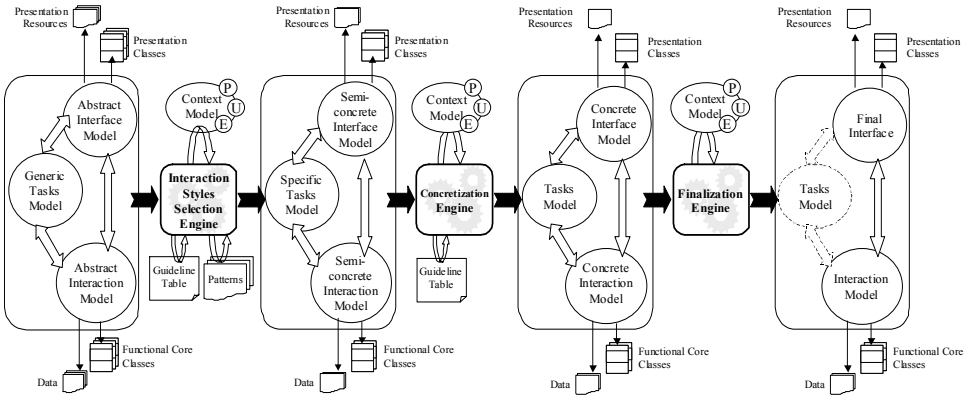


Figure 1.

Our vision of the complete process of MUI generation

BASIC DESCRIPTION OF AMF

A large number of architectures for Interactive Software have been described, e.g., MVC (Model-View-Controller) [4], PAC (Presentation-Abstraction-Controller) [3], ADC (Abstraction-Display-Controller) [6]. Most of these architectures are based on the traditional view of interactive software, namely the view that an interactive software system can be separated into the application and the user interface. The application part contains the functionality of the software and the user interface part contains the representation of this functionality proposed to the application user(s).

AMF is a multi-agents and multi-facets architecture model that specifies the software architecture of an interactive application. It enables the design of reusable elements. It can be extended and adapted to the need of specific applications. The AMF model can be seen as an upgrade of the PAC and MVC models. It combines the conceptual powerfulness of multi-agents architectures such as PAC while providing an operational implementation schema, which is a key factor of the success of MVC.

Fundamentally, AMF provides four key features:

- It generalizes the concept of facet, extending their number from 3 in both MVC and PAC to n, i.e. an open-ended set of useful facets (e.g. Cooperation in CSCW);
- It formalizes the control components;
- It fits well with task modelling approaches and design patterns;
- It defines an API and relies on a powerful runnable engine.

AMF provides a graphical formalism that represents the structure and specifies the temporal sequence of processes. Finally, a Java implementation of an AMF engine enables the execution of an AMF model coupled to applicative classes.

The class 'agent' is the basic component of AMF models. Each agent is made of facets and control administrators. It can imply other agents. Each class agent can generate several instances. Each facet incorporates logical communication ports and is associated to an applicative class where some functions, called «daemons», are mapped to the ports.

AMF proposes a unified formalism to model control components because such formalisms are rare and usually difficult to use in real contexts (see Petri nets for instance). Yet, these components are the major pieces of architectural models and it is of great importance to provide an efficient modelling tool. The control component of each agent is its main part because it manages all the communications between the facets of the agent and other agents. AMF defines 2 kinds of elements:

- At the Facet level, communication ports present the services that are offered by the facet and the ones that are needed (respectively input and output ports).
- At the Agent level, control administrators are connecting communication ports. These administrators can easily be standardized (OR, AND, Sequence, etc.) and extended to handle complex controls such as multi-user synchronization or interaction tracking.

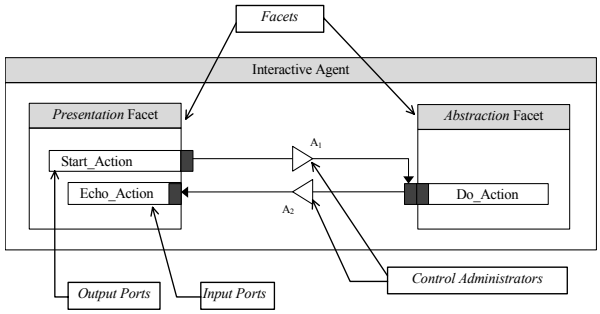


Figure 2. Basic elements of AMF architecture

We briefly introduce 2 special features of the control administrators:

- After being activated, a target port is always returning a message to the source port. This "acknowledgement" message is generally ignored but it can be used to return data to the source port. When it is the case, the control administrator is represented with a black triangle (see figure 3a).
- The possible existence of multiple instances of a unique class drove us to provide a default mechanism that broadcasts messages from a control administrator to the target ports of all the instances of an agent. To be able to activate a specific instance of an agent, we add an optional parameter to the activate function in order to explicitly define a target agent. The identity of the agent is usually known only during runtime. So we do not need a new type of administrator but only a new activation technique. Yet, for a better understanding of the visual model, our advice is to add a little vertical bar at the end of the administrator to explain that a filtering is done on the target agents (see figure 3b).

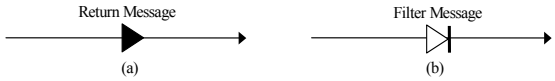


Figure 3. Return and Filter features of control administrators.

Finally, a Java implementation of an AMF engine enables the execution of an AMF model coupled to applicative classes.

The AMF Model can be published using an XML notation. Here is the Document Type Definition we use:

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT Agent (Agent*, Administrator+, Facet+)>
<!ATTLIST Agent
    Name CDATA #REQUIRED
    Sub-agent CDATA #REQUIRED
    Type CDATA #REQUIRED
>
<!ELEMENT Administrator (Sources+, Targets+)>
<!ATTLIST Administrator
    Name CDATA #REQUIRED
    Type (Simple | Return | Filter | ReturnFilter | Sequence)
#REQUIRED
    TypeAC (Abstract | Concrete) #REQUIRED
>
<!ELEMENT Targets EMPTY>
<!ATTLIST Targets
    Name CDATA #REQUIRED
>
<!ELEMENT Sources EMPTY>
<!ATTLIST Sources
    Name CDATA #REQUIRED
>
<!ELEMENT Facet (Port+)>
<!ATTLIST Facet
    Name CDATA #REQUIRED
    Type CDATA #REQUIRED
>
<!ELEMENT Port EMPTY>
<!ATTLIST Port
    Name CDATA #IMPLIED
    Type CDATA #REQUIRED
    TypeIO (2 | i | o) #REQUIRED
    TypeAC (Abstract | Concrete) #REQUIRED
    DaemonName CDATA #REQUIRED
    FacetName CDATA #REQUIRED
>
```

The AMF Engine

The goals of the AMF are to help design, implementation, use and maintenance. Our approach consists in combining both multiagent view (like PAC) and layered view (like Arch). The multiagent view is used during the design and a layered technology is used for implementation.

Actually, agents are dual entities: one part located into the AMF engine manages the control of the interactions while another one, on the application side, manages both widgets for interactivity and real domain-dependent abstractions.

The 5 levels of Arch model are present:

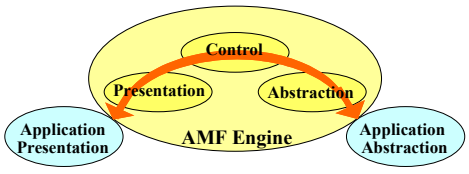


Figure 4. The Layers of the AMF implementation

To implement AMF architecture, we built an engine that manages all the AMF objects (agents, facets, ports and administrators) and their communications. The external elements, which are both objects that define the functional kernel of the application and objects that use a graphical toolkit, are linked to the AMF objects. For instance, each communication port is associated to a function called daemon in the Application side. This daemon is automatically triggered when the port is activated.

At runtime, for each user's action (button pressed, menu selection…), the corresponding event received by an application object (i.e. the one that manages the window) activates an output port of the associated AMF agent in the engine (↘ symbol in the graphical models). At the end of the control processing, input ports are activated and their daemons are run.

AMF concepts can be compared to ones of Java Beans. Indeed, facets are components (Beans) that are able to present themselves (with their ports) and that communicate by sending and receiving messages. Ports and administrators are very similar to listeners and adapters (in fact, the Java implementation of AMF uses them). However, AMF relies on a sophisticated engine so that programmers can use predefined components, such as standard administrators, which are real objects and not only java interfaces.

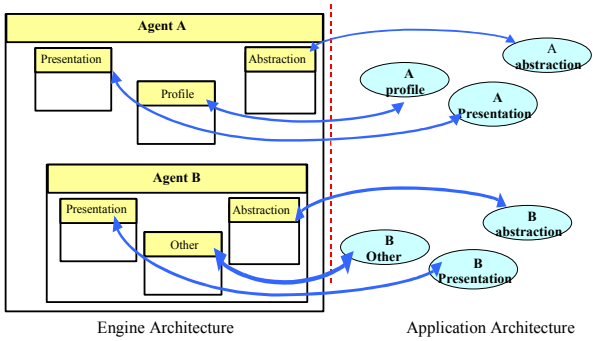


Figure 5. Links between AMF objects inside the engine and application classes outside.

THE AMF DESCRIPTION OF AN ABSTRACT INTERACTION MODEL

To illustrate our approach, we are considering a classical game called «The Towers of Hanoi » (figure 6), which consists in moving rings of different sizes to reach a goal. The rings are stacked up on three stems; they have an initial position and should be moved to reach a target-position. The shifting must respect the following rules: only one ring can be moved at a time and a ring with a given size cannot be placed upon a ring of a smaller size.

There are three types of object in this application: the game which contains the rule and the other objects, the stem (with three instances) upon which the rings are slipped and the ring (with 3 to 5 instances according to the complexity of the game). The interaction consists in a succession of operations: the selection of the ring (on the top of a stem) followed by the shifting, then the validation

of the move (respect rule) and finally, the detection of the end of the game.

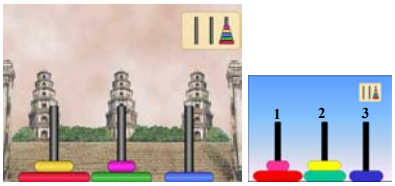


Figure 6. "The Towers of Hanoi" application

The first step for the designer consists in defining a generic task model. The Task Model is a tasks tree that is hierarchically organized. Various formalisms have been proposed to model the task model. We use the CTT notation and CTTE editor [9] for its description and modeling. In our approach, the Generic task model contains regular nodes corresponding to common tasks and abstract nodes that will be "specialized" later in relation with the context of use (e.g. a "Selection&Move" node that will be specialized by a "Drag & Drop" sub-tree). Figure 7 presents the Generic task model of "The Towers of Hanoi" using a CTT notation.

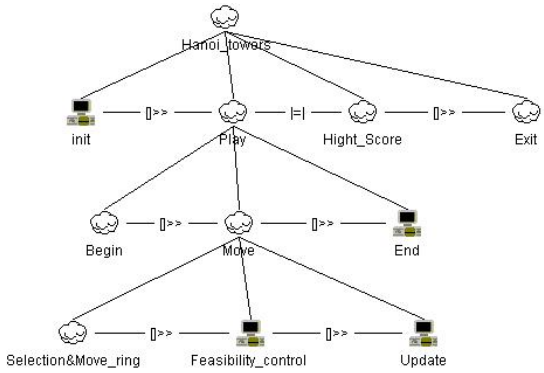


Figure 7. Generic Task Model of "The Towers of Hanoi"

From the generic task model we can establish the abstract interaction model of the application. This model is an AMF description that contains abstract ports. These ports represent functionalities that can be executed differently according to the specifications of the target.

Figure 8 represents the abstract interaction model of "The Towers of Hanoi" game. After a move, if it is a valid one (*Validate_move ports*), the scene must be re-painted (*Refresh ports*). The task of selecting and moving a ring in this model is abstract (elements are represented with dotted lines). Indeed, this action can be carried out differently according to the means of interaction that are available on the given target: a mouse, then the user may drag & drop, or a keyboard, then he/she will type the number of the source stem and after the target stem's one. To skip from an abstract interaction model to a concrete model, we need to replace abstract ports using the patterns.

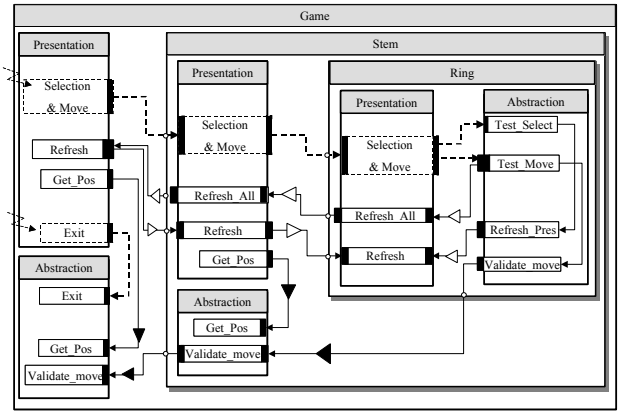


Figure 8. Abstract Interaction Model of "The Towers of Hanoi"

Here is an extract of the XML description of the AMF abstract model for the "Towers of Hanoi" application. We only detail the Ring agent. Note that the "selection_move" port is an abstract port. In addition, the names of the elements are rich ("#" symbols are used by the engine) so that we can use dynamic links.

```
<?xml version="1.0" encoding="UTF_8"?>
<!DOCTYPE Agent SYSTEM "amf.dtd">
<Agent Name="GAME" Sub-agent="1" Type="game">
    <Agent Name="STEM" Sub-agent="1" Type="stem">
        <Agent Name="RING" Sub-agent="0" Type="ring">
            …
            <Administrator
            Name="Test_Select#RING#STEM#GAME"
            Type="Return" TypeAC="Abstract">
                <Sources
        Name="Selection_Move#PRESENT#RING#STEM#GAME"/>
                <Targets
            Name="Test_Select#ABSTR#RING#STEM#GAME"/>
            </Administrator>
            <Facet Name="ABSTR#RING#STEM#GAME"
            Type="abstr#ring#stem#game">
                <Port
            Name="REFRESH#ABSTR#RING#STEM#GAME"
        Type="refresh#abstr#ring#stem#game" TypeIO="2"
            TypeAC="Concrete"
            FacetName="ABSTR#RING#STEM#GAME"
            DaemonName="refresh"/>
                <Port
            Name="TEST_MOVE#ABSTR#RING#STEM#GAME"
            Type="test_move#abstr#ring#stem#game" TypeIO="2"
            TypeAC="Concrete"
            FacetName="ABSTR#RING#STEM#GAME"
            DaemonName="test_move"/>
                …
                <Port
            Name="Refresh_all#PRESENT#RING#STEM#GAME"
            Type="refresh_all#present#ring#stem#game" TypeIO="2"
            TypeAC="Concrete"
            FacetName="PRESENT#RING#STEM#GAME"
            DaemonName="refresh_all"/>
                <Port
        Name="Selection_Move#PRESENT#RING#STEM#GAME"
            Type="selection_move#present#ring#stem#game"
            TypeIO="1" TypeAC="Abstract"
```

```
            FacetName="PRESENT#RING#STEM#GAME"
            DaemonName="null"/>
            </Facet>
        </Agent>
        …
</Agent>
```

THE INTERACTION PATTERNS

The AMF model is a part of the «design patterns» approach because some combinations of agents – facets – ports constitute potential patterns that can be isolated and described.

Thus, we have defined several patterns related to interaction means (mouse, keyboard…) that are used to interact with the application in different contexts. For sure, other patterns may be defined.

As an example we present hereafter an interaction pattern used for a task of selection and removal of an element among a set of elements located into a container (Fig 9). This pattern is applied if the interaction is done with a mouse.

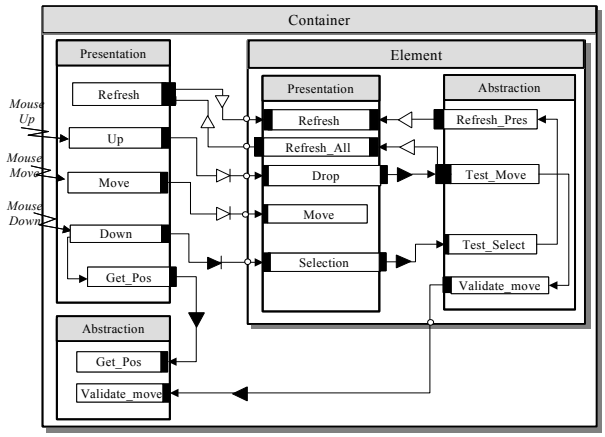


Figure 9. The graphical "Select and Move" Pattern for a mouse.

For this pattern we have the following XML description:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Agent SYSTEM "amf.dtd">
<Agent Name="CONTAINER" Sub-agent="1" Type="container">
    <Agent Name="ELEMENT" Sub-agent="0" Type="element">
        <Administrator Name="Refresh#ELEMENT#CONTAINER"
        Type="Simple" TypeAC="Concrete">
            <Sources
        Name="REFRESH#ABSTR#ELEMENT#CONTAINER"/>
            <Targets
        Name="REFRESH#PRESENT#ELEMENT#CONTAINER"/>
        </Administrator>
        …
        <Facet Name="PRESENT#ELEMENT#CONTAINER"
        Type="present#element#container">
        …
            <Port
            Name="MOVE#PRESENT#ELEMENT#CONTAINER"
            Type="move#present#element#container" TypeIO="i"
            TypeAC="Concrete"
            FacetName="PRESENT#ELEMENT#CONTAINER"
            DaemonName="Move"/>
            <Port
        Name="SELECTION#PRESENT#ELEMENT#CONTAINER"
            Type="selection#present#element#container" TypeIO="2"
            TypeAC="Concrete"
            FacetName="PRESENT#ELEMENT#CONTAINER"
            DaemonName="Selection"/>
        </Facet>
    </Agent>
    <Administrator Name="Refresh#CONTAINER" Type="Simple"
    TypeAC="Concrete">
        …
    <Administrator Name="Move#CONTAINER" Type="Filter"
    TypeAC="Concrete">
        <Sources Name="MOVE#PRESENT#CONTAINER"/>
        <Targets
        Name="MOVE#PRESENT#ELEMENT#CONTAINER"/>
    </Administrator>
    <Administrator Name="Drop#CONTAINER" Type="Filter"
    TypeAC="Concrete">
        <Sources Name="UP#PRESENT#CONTAINER"/>
        <Targets
        Name="DROP#PRESENT#ELEMENT#CONTAINER"/>
    </Administrator>
    <Administrator Name="Selection#CONTAINER"
    Type="ReturnFilter" TypeAC="Concrete">
        <Sources Name="DOWN#PRESENT#CONTAINER"/>
        <Targets
    Name="SELECTION#PRESENT#ELEMENT#CONTAINER"/>
    </Administrator>
    …
    <Facet Name="ABSTR#CONTAINER" Type="abstr#container">
        <Port Name="GET_POS#ABSTR#CONTAINER"
        Type="get_pos#abstr#container" TypeIO="i"
        TypeAC="Concrete"  FacetName="ABSTR#CONTAINER"
        DaemonName="Get_pos"/>
        <Port Name="VALIDATE#ABSTR#CONTAINER"
        Type="validate#abstr#container" TypeIO="2"
        TypeAC="Concrete"  FacetName="ABSTR#CONTAINER"
        DaemonName="Validate"/>
    </Facet>
    …
</Agent>
```

INTERACTION MODELS ADAPTATION

The adaptation engine replaces the abstract tasks with a concrete task and the interaction pattern that is related to the task is inserted into the abstract interaction model of the application. This replacement is done according to the characteristics of the target. Hence, concrete ports and concrete administrators will replace the abstract ports that are inside the abstract interaction model.

A name-based approach is used to replace the generic names (CONTAINER & ELEMENT) by the concrete ones (STEM & RING).

If we consider the Towers of Hanoi example running on a platform with a mouse, the pattern presented in the figure 9 will be instanced. A name-based rule enables to maintain the link between the ports and the interface element that receives the action. Then, the pattern replaces the abstract ports in the interaction model of the application. This process produces a final interaction model of the application (figure 10).

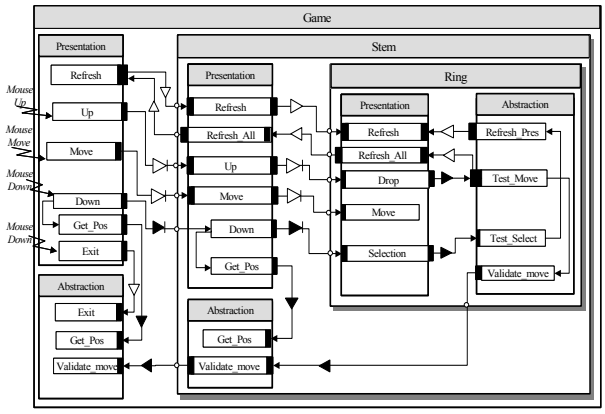


Figure 10. Concrete interaction model of the Towers of Hanoi with a mouse

Here is an extract of the XML description of the AMF concrete model for the application. The DaemonName fields of the concrete ports are method names of the Java classes defined by the developer.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Agent SYSTEM "amf.dtd">
<Agent Name="GAME" Sub-agent="1" Type="game">
    <Agent Name="STEM" Sub-agent="1" Type="stem">
        <Agent Name="RING" Sub-agent="0" Type="ring">
            <Administrator Name="Refresh#RING#STEM#GAME"
            Type="Simple" TypeAC="Concrete">
                <Sources
            Name="Refresh#ABSTR#RING#STEM#GAME"/>
                <Targets
            Name="Refresh#PRESENT#RING#STEM#GAME"/>
            </Administrator>
            <Administrator
            Name="Refresh_all#RING#STEM#GAME"
            Type="Simple" TypeAC="Concrete">
                <Sources
            Name="Test_Move#ABSTR#RING#STEM#GAME"/>
                <Targets
            Name="Refresh_all#PRESENT#RING#STEM#GAME"/>
            </Administrator>
            …

                <Port
            Name="DROP#PRESENT#RING#STEM#GAME"
            Type="drop#present#ring#stem#game" TypeIO="2"
            TypeAC="Concrete"
            FacetName="PRESENT#RING#STEM#GAME"
            DaemonName="Drop"/>
                <Port
            Name="MOVE#PRESENT#RING#STEM#GAME"
            Type="move#present#ring#stem#game" TypeIO="i"
            TypeAC="Concrete"
            FacetName="PRESENT#RING#STEM#GAME"
            DaemonName="Move"/>
                <Port
            Name="SELECTION#PRESENT#RING#STEM#GAME"
            Type="selection#present#ring#stem#game" TypeIO="2"
            TypeAC="Concrete"
            FacetName="PRESENT#RING#STEM#GAME"
            DaemonName="Selection"/>
            </Facet>
        </Agent>
…
    </Agent>
…
</Agent>
```

The DaemonName fields of the concrete ports are method names of the Java classes defined by the developer. Here is the interface of the RingPres class.

```
// File : iRingPres.java
public interface iRingPres
{
        void    Refresh();
        void    Refresh_All();
        private Down(MouseEvent arg);
        private Move(MouseEvent arg);
        private Drop(MouseEvent arg);
}
// iRingPres
```

CONCLUSION

In this paper we have shortly presented an architecture-based approach for the generation of Multiple User Interfaces. It incorporates the use of task patterns and interaction patterns. To describe the interaction model and the interaction patterns we used the AMF architecture which is composed of an XML description of an AMF model and a run-time engine.

We use AMF to define the interaction model, which enables us to obtain an abstract description of the interaction model of the application. Processing (filtering and enriching) this description with the XML parsing mechanisms helps us to concretise the abstract model in a progressive way. At the end of the process, we obtain a concrete description of the AMF interaction model. A Java implementation of an AMF engine enables the execution of an AMF model coupled to applicative classes. We can now imagine building other AMF players (non-java) that will allow the application to the AMF-XML files.

This approach is original in the sense that it tries to unify the task model, the interaction model and the resources of the application, i.e. the functional resources (Java classes of the application domain) and interaction resources (images, menus…).

The designer has to specify the task model, the interaction model, the java classes (which ensure the various interaction styles) and the presentation resources. The system analyses these elements and, using interaction guidelines and patterns, it maps filtered elements on the resources.

We are aware of the difficulties and limits in considering the definition of a process that is wholly automatic. The complexity of the problem requires simplifications that inevitably lead to stereotyped and non-adapted interfaces to the specificity of materials. The introduction of the adapted task patterns and the interaction ones may decrease the complexity of the issue. However, it is obvious that the contribution of the designer should take place in this type of process. In this context, we will consider the introduction of a constraint definition file. The designer defines this file, which is used to restrict the modifications upon some elements during the process of the dynamic generation of the concrete interface.

REFERENCES

1. Abrams M., Phanouriou C., Baeongbacal A. L., Williams S. M., Shuster J.E., "UIML: An Appliance-Independent XML User Interface Language," In Computer Networks, Vol. 31, 1999, pp. 1695-1708.
2. Calvary G., Coutaz J., Thevenin D., Limbourg Q., Bouillon L., Vanderdonckt J. A unifying reference framework for multi-target user interfaces, Journal of Interacting With Computer, Elsevier Science B.V, June, 2003, Vol 15/3, pp 289-308.
3. Coutaz J.: PAC, an Object Oriented Model for Dialog Design, in Proceedings Interact'87, North Holland, 1987, pp.431-436.
4. Krasner G.E., Pope S.T. A Cookbook For Using the Model-View-Controller User Interface Paradigm in The Smalltalk-80 System. Journal of Object Oriented Programming, 1988, 1, 3, pp. 26-49.
5. Luyten K., Van Laerhoven T., Coninx K., Van Reeth F., «Runtime transformations for modal independent user interface migration». Interacting with Computers. Vol. 15, No. 3, June 2003. pp. 329–347.
6. Markopoulos P, Johnson P. Rowson J. Formal architectural abstractions for interactive software. International Journal of Human Computer Studies, Academic Press, (1998), 49, pp. 679-715.
7. Mori G., Paternò F., Santoro C. « Tool Support for Designing Nomadic Applications» Proceedings of IUI 2003, Miami, Florida, January 12-15, 2003.
8. Paternò F., Model-based Design and Evaluation of Interactive Applications. Springer-Verlag, November 1999.
9. Seffah, A., Radhakrishan T., Canals G. Workshop on Multiples User Interfaces over the Internet: Engineering and Applications Trends. IHM-HCI: French/British Conference on Human Computer Interaction, September 10-14, 2001, Lille, France.
10. Samaan K., Tarpin-Bernard F. « L'Utilisation de Patterns d'Interaction pour l'Adaptation d'IHM Multicibles ». IHM'03, CAEN-FRANCE, novembre 2003.
11. Souchon N., Limbourg Q., Vanderdonckt J. Task Modeling in Multiple Contexts of Use. In Proceedings of DSVIS'2002 Workshop. 2002.
12. Tarpin-Bernard F., David B.T., *AMF : un modèle d'architecture multi-agents multi-facettes.* Techniques et Sciences Informatiques. Hermès. Paris. Vol. 18. No. 5. p. 555-586. Mai 1999.
13. Thevenin, D., Coutaz, J. Plasticity of User Interfaces: Framework and Research Agenda. In Proceedings of INTERACT'99, 1999, pp. 110-117.
14. Vanderdonckt, J., Bodart, F., 1993. Encapsulating knowledge for intelligent automatic interaction objects selection. In: Ashlund, S., Mullet, K., Henderson, A., Hollnagel, E., White, T. (Eds.), Proceedings of the ACM Conference on Human Factors in Computing Systems InterCHI'93 Amsterdam, 24–29 April 1993), ACM Press, New York, pp. 424–429.
15. XIML Forum Site Web. http://www.ximl.org.

Supporting Workflow in User Interface Description Languages

Nicole Stavness
Department of Computer Science
University of Saskatchewan
Saskatoon, Saskatchewan, Canada
+1 306 966 8654
nicole.stavness@usask.ca

Kevin Schneider
Department of Computer Science
University of Saskatchewan
Saskatoon, Saskatchewan, Canada
+1 306 966 4891
kas@cs.usask.ca

ABSTRACT

XML-based user interface description languages (UIDLs) have been developed to support user interface portability across multiple platforms. UIDLs express various aspects of the user interface, including the abstract and concrete elements of the user interface, the tasks to be performed by the user, and the user interface dialogue.

We have developed the progression model for expressing workflow aspects of an interactive system using an XML-based language. The progression model considers workflow to be a sequence of *scenes* progressing towards an organizational goal. The model allows us to express workflow explicitly using a markup language.

In this paper we present a prototype system, the progression analyzer that accepts a progression, renders the user interface described by a *scene* and provides the user with a mechanism to monitor, save, recall, reorder and coordinate the workflow.

Keywords

Workflow, Task Model, User Interface Description Language, XML

INTRODUCTION

Business organizations use interactive information systems to support their business processes. Users require flexibility when interacting with the system to contend with changes in the business processes, to support differing work approaches, and to coordinate the activities of various workers. This flexibility can be supported by workflow systems.

UIDLs specify important aspects of the user interface. Unfortunately, UIDLs do not express aspects of the user interface related to workflow. We have developed the progression model [18] to explicitly express workflow as a sequence of *scenes* called a *progression*. Our language to express progressions is XML-based.

Our approach is specifically aimed at the development of transaction-based interactive systems. A progression in this context may be renting a car, filling out a mortgage application, or booking a flight.

In this paper we present a prototype system, the progression analyzer, for rendering and interacting with progressions. After a discussion of the related work and introducing the progression model we describe the progression analyzer prototype. We then evaluate our approach by showing how it maps to the components commonly found in a workflow system, using examples from our prototype. We conclude the paper with a discussion of future research directions.

RELATED WORK

In this section we describe workflow models, task models and how task and workflow models differ with respect to user interface development. As well we briefly describe some current XML-compliant UIDLs.

Workflow Model

The workflow model is used to represent the flow of work within a department, across a company or to external agents [14]. The Workflow Management Coalition (WFMC) defines workflow as the automation of a business process, in whole or part, during which documents, information, or tasks are passed from one participant to another for action, according to a set of procedural rules [20]. The details of a specific business process are defined in a process definition. This includes the sequences of activities and associated relationships; start and finish criteria; and information such as executers of manual or automated activities, procedural rules, and control data. The process definition may also include sub-process descriptions.

Workflow research focuses on approaches to making changes during the business process. Procedure-like, routine processes that are statically supported are on one end of a continuum, and highly unspecified, dynamic processes are on the other end [5]. In adaptive workflow

management systems [13] the procedural rules can be changed or created during the process. Some research proposes a cooperative hypermedia system, with process support through a meta-model, to integrate the efforts towards communication, coordination, and cooperation in workflow systems [7]. A two-part classification defines types of possible flexibilities that may be desired in workflow management applications [9]. 'Flexibility by selection' provides the user some leniency in executing a process by offering multiple execution paths. Alternatively, 'flexibility by adaption' provides the ability to add extra execution paths through additional functionality and tools that allow the workflow type to change and integrate during runtime.

Some workflow concepts that are common in many systems have been identified in [12]. This research has focused on applying workflow to object oriented systems; the following concepts are identified as important to workflow.

- *Monitoring* for contributing information about the circumstances of workflows during execution;
- *History* of workflow actions for evaluation or recovery;
- *Persistence* to save the historic information and provide access to it;
- *Manual Intervention* for changing the order that activities are performed in as they are performed;
- *Worklist* to coordinate the activities among the workers;
- *Federated Workflow* addresses the issue of how workflow systems interoperate.

Task Model

Task Models are logical descriptions of activities that are designed to be carried out in reaching user's goals in an interactive system. There are many different approaches to task modeling such as Hierarchical Task Analysis [2], GOMS [6], UAN [8], and ConcurTaskTrees [14].

Hierarchical Task Analysis (HTA) is based on describing the set of goals, tasks and operations in logical structures of different levels. GOMS (goals, operators, methods, selection) depicts procedural knowledge or 'how-to-do-it" knowledge through fine-grained operators that are performed to reach a goal. UAN (User Action Notation) also follows a hierarchical structure. It provides a notation for designers to describe the dynamic behaviour of graphical user interfaces, where the tasks are represented asynchronously with operators that denote the temporal relationships. The ConcurTaskTrees notation was created to support engineering approaches to task modeling. Temporal relationships are also incorporated for enabling, concurrency, disabling, interruption, and optionality. Additionally, synchronized tasks where the output information of one task is the input information of another are supported.

In relation to business processes, task models describe the paths of activities available to reach the user's goals. Unfortunately, task models often result in large specifications with more detail than is necessary for a designer. Recent research has investigated annotating task models with data artifacts to better support information systems and extracting dialog models from the task model to better support automated generation of user interfaces [10].

Workflow models and a task models both describe how to accomplish work or tasks. As identified in [19], they both have similar concepts, such as actions/tasks and workers/users. Alternatively, [19] points out that workflow models are useful for group or organization interaction, while task models focus on individual users. This coincides with our research that associates workflow models with focusing on the management of task accomplishment processes. In groups or organizations more direction is required to ensure that orderliness and goal accomplishment are maintained. Workflow research as discussed in [12] goes beyond the actual task and provides a meta-level that focuses on how to coordinate the activities towards completion and how to examine them thereafter.

User Interface Description Languages

Souchon and Vanderdonckt [17] have analyzed a number of XML-compliant languages for defining user interfaces including UIML [1], AUIML [4], XIML [16], Seescoa XML [11], Teresa XML [15], and WSXL [3].

User Interface Markup Language (UIML) allows the user to specify the user interface in general terms then render it according to a style description. Abstract User Interface Markup Language (AUIML) focuses on describing the desired user interaction in terms of its purpose rather than appearance. The eXtensible Interface Markup Language (XIML) affords the ability to describe a user interface without concern for the implementation. Software Engineering for Embedded Systems using a Component Oriented Approach (Seescoa XML) defines an XML description to express an abstraction of the user interface using Java User Interface components. Teresa XML provides a facility to support the design and generate a concrete user interface for a specific type of platform. Web Services Experience Language (WSXL) focuses on a web services model to interact with web applications. The User Interface Description Languages referenced above do not address workflow issues.

PROGRESSION MODEL

The progression model [18] incorporates workflow features into a markup language specification. This research is not concerned with the actual rendering of the user interface as is addressed in many other UIDLs. The progression model makes explicit the steps and transactions a user makes when using a transaction–based information system. As the user progresses towards accomplishing a task or goal, the

progression model infrastructure records each step and the state of the transaction.

Making the steps and transactions explicit allows the user to group transactions into batches for later processing, to store partial transactions for later editing, and allows the user to browse historical progressions. Linking the steps in the workflow directly to the transaction provides a means to integrate the process model and the data model in one coherent model. This enables the support of the flow of work for an individual user by supporting new interactions. A series of definitions outline the basic aspects of a progression. Consequently, new interactions are enabled to provide flexible business process support.

Definitions

The following definitions describe the key elements of the progression model and how they relate to each other. These items are graphically depicted in Figure 1.

Progression. A *progression*, p, is a sequence of scenes (or steps), s, in a process to create a transaction, that is $p = < s_1, \ldots, s_n >$.

Progression Interval. A *progression interval*, pi, is a subsequence of a progression or a couple steps.

Scene. A *scene*, s, corresponds to a step in a progression. Each scene of a progression is associated with the user interface, u, current state of the transaction, t, and current state of the workflow, w, therefore, $s = < u, t, w, >$. A scene captures the process and associated data as a user performs actions throughout a progression.

User Interface. The *user interface*, u, is a rendering of the user interface for the current scene. The user can perform user actions, a, according to the components, such as a text field or select box, available in the user interface.

User Actions. The *user actions*, a, for a scene are the interactions that the user performs within the user interface.

Transaction. The *transaction*, t, models the accumulation of information at each point in the progression. Each transaction is made up of a series of *elements*, e, that are accumulated throughout the progression by user actions, that is $t = < e_i, \ldots, e_n >$. As the scenes change, the element additions, deletions, and changes are reflected in the transaction. For instance, if the user is filling out a wizard form, at every submission the new information is added to the transaction. A series of zero or more *user actions*, a, can be performed directly on the transaction. For example, the user may want to directly edit a field in the transaction rather than going back and editing through the user interface.

Workflow. The *workflow*, w, is a sequence of scenes progressing towards an organizational goal. It identifies the scenes that are completed, currently in progress, and not yet started. It also defines who is assigned to complete a scene. Additionally, it outlines the available *workflow actions*, wa, for the current scene.

Workflow Action. A *workflow action*, wa, is an action that affects the workflow of the progression. One type of workflow action is "transform", which may send information to the transaction and change the user interface to a new scene. For example, when the user clicks on a submit button, a new scene is generated. The information that was entered in the previous scene, such as the text entered in a form is reflected in the transaction. The feedback is then displayed to the user through the user interface. The other type of workflow action deals with interacting with the progression. For example, the user can recall a past progression, replay a progression, save a progression, and so on.

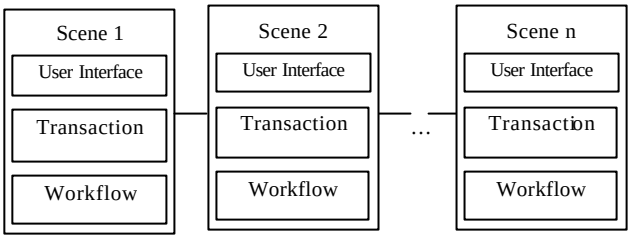


Figure 1. A progression is a sequence of scenes. The rectangles represent scenes that encompass a user interface, a transaction, and workflow.

Benefits of Recording Progressions

An information system is developed to support an organization's business processes. This requires a high degree of flexibility, which has been traditionally difficult to support. The process and data information that is captured through the progression model can be used to support flexible business processes. It is facilitated by displaying the transaction to the user, in addition to the original interface, accompanied by new functionality. Through opening the model to the user in this way, a number of new interactions become available to the user. The interactions that are enabled include: information orientation, immediate updates, historical review, concurrent process comparison, progression batching, and progression manipulation.

Progression Orientation

By visually observing the transaction, the user is able to see the information being built up while the progression is enacted. This provides a reference for the user to ensure the information is correct. Additionally, foresight into the information that is required later in the progression is available from the beginning. This allows users to organize and anticipate the work required to complete the progression. Users that are new to the system now have the ability to reduce the unknown aspects of the system.

Progression Updates

Direct editing of the accumulated data is available while enacting a progression. A user can change information at

any time without having to go backward in the progression and forfeit the later information, such as in web browsing. This also allows the user to keep track of their placement within the progression. Updates may not be allowed for some information items as the constraints of the system must be upheld. Nonetheless, flexibility to make direct changes to the information already accumulated is afforded.

Progression History

A user has access to the progression history. History includes the progression scenes, as well as the transaction snapshots. The user can benefit from the ability to change, replay, or reuse historical information. Changing the history allows the user to move backward in the progression to undo actions. Replaying a progression may be useful for learning how progressions were previously completed by others; remembering what the user did last time they went through the progression; or for the supervisor to look at the work that an employee has performed. New or infrequent users would find the most benefit from this interaction. It is also useful for lengthy progressions, to view work that is not easily remembered. By saving the history of the progression, partial progressions can be closed and returned to at a later time or parts of saved progressions can be reused in future progressions. This is useful when the user is interrupted during a session before they can complete the progression. Alternatively, when the work requirements are more ambiguous and combinations of different progressions are useful.

Multiple Progressions

Multiple progressions can be used to process transactions concurrently. The ability to duplicate and/or view more than one progression at the same time allows for easy comparisons without having to lose work that is already completed, such as when trying out different scenarios or outcomes. The user can go through one possibility, then without losing that information try out another scenario. The outcomes can be considered in a side by side manner.

Progression Batches

Progressions can be applied to multiple items to enable the user to perform a progression and have it affect more than one selected item in the system. For example, a user can perform a progression to change an employee's salary, but have it apply to ten employees. This is beneficial for saving time and consistency while managing large amounts of information.

PROGRESSION ANALYZER

The progression analyzer is a prototype system for displaying information about a progression. A progression is modeled using a mark-up language. Figure 2 shows the outline of the progression model in the markup format. Each scene consists of an abstract user interface (aui) with the corresponding user actions; a set of transactions with possible transaction actions; and finally the workflow with available workflow actions.

The parsed code is interpreted and explicit sections are depicted in the progression analyzer interface. Figure 3 shows a screenshot of the prototype without content.

```
<progression>
    <scene>
        <aui>…</aui>
        <transactions>
            <transaction>…</transaction>
        </transactions>
        <workflow> … </workflow>
    </scene>
    …
</progression>
```

Figure 2. Skeleton of mark-up language for the progression model.

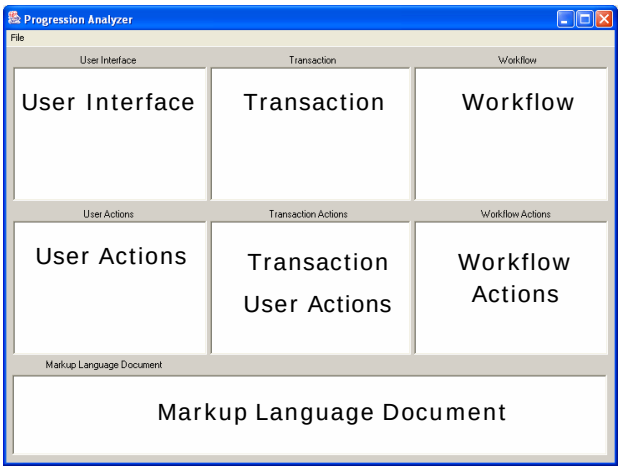


Figure 3. An outline of the progression analyzer panels .

The user interface is rendered in the user interface panel. From the user interface information, the specific user actions are extracted and displayed in the user actions panel. The transaction is shown in a table indicating the transaction number, transaction structure, and the status of the transaction. Any transaction actions that are permissible, such as directly edit field, are presented in the transaction actions panel. The workflow is presented as a table with the scene number, scene name, worker assigned to complete the scene, and status of the workflow scene. Then the possible workflow actions, such as reorder and history, are displayed in the workflow actions panel. Additionally, the mark-up language document is displayed in the corresponding panel.

When the user selects the "transform" workflow action to create and display the next scene, the markup language for that scene is derived from the previous scene and the user actions. The new scene is added to the markup language document and displayed in the progression analyzer panels.

User Interface

The user interface panel shows the rendering of the user interface as it appears to the user. It is non-editable and intended to show the user the snapshot of the user interface at the beginning of the current scene.

User Actions

The user actions panel shows the user all the possible actions that are available to perform within the user interface. The user can interact with the elements on this form and perform user actions toward the progression. For example, the user can enter some text in a text field or select from a select box and so on.

Transaction

The transaction panel depicts the transaction in a table. The transaction consists of all the required information that is accumulated throughout the progression to successfully submit and complete the progression. Each transaction is numbered for unique identification. There is also a status field to indicate the current transaction state, such as partial, complete, valid, or invalid. The remaining fields depend on the information requirements of the transaction.

Transaction Actions

The transaction actions panel displays the available actions that the user can perform on the transaction structure itself. For example, the user may want to edit a field in the transaction directly rather than going through the user interface. System constraints may restrict the user from performing some transaction actions.

Workflow

The workflow panel shows the scenes that are completed during a progression. The workflow panel lays out to the user the scenes that are completed, the scene to be completed next, and the scenes that still need to be completed. Each scene is numbered for unique identification. The status column provides the information on whether the scene is completed, currently being worked on, or yet to be completed. The worker column indicates which human user is assigned to complete the scene actions. This could also be extended to include jobs that the system must complete to show the interaction with the application.

Workflow Actions

The workflow actions panel shows all the possible workflow actions that the user can perform during the interaction with the user interface. Some workflow actions that we have identified as interesting are: move to the previous scene; move to the next scene; reorder the scenes within the limits of system constraints; transform the scene as the user actions are complete; add the transaction to a batch; save the partially completed progression; and view the progression history.

THE PROGRESSION ANALYZER AND WORKFLOW

Manolescu's research [12] maps workflow concepts to an object oriented framework. In his research he identified six components that are common to workflow systems, namely: monitoring, history, persistence, manual intervention, worklist and federated workflow. In this section a comparison is made between each of these workflow components and the progression model using examples from our prototype.

Monitoring refers to gathering information on the state of the workflow regarding the progress of the activities within the workflow. In the progression model, this relates to presenting the user with information on what work has been done, what is currently being done, and what work remains to be done.

In the example depicted in Figure 4 and Figure 5, the first four scenes are completed in the progression and four remain uncompleted. The worker named Jen is responsible for completing the last two scenes of the progression. She wants to see how far along the other workers are in completing the scenes. By viewing the workflow panel, as shown in Figure 4, she can see that the first four scenes are completed and the fifth is in progress.

Workflow

Scene	Name	Worker	Status
1	Name	Tara	Complete
2	Address	Tara	Complete
3	Account	Jim	Complete
4	Billing	Mark	Complete
5	Shipping	Dave	In Progress
6	Approved	Mark	Inactive
7	Credit	Jen	Inactive
8	Confirm	Jen	Inactive

Figure 4. The workflow panel depicting the status of the workflow.

Additionally, when she is completing her scenes, she can view the buildup of information in the transaction. In Figure 5 she has just entered the address information and then selected the transform workflow action. When she looks at the transaction in third scene, she can see the information that resulted from her actions in the previous scene through the transaction panel. Her address information is added to the correct fields in the transaction.

History refers to recording the actions that were taken during the execution of the workflow. This can be used for evaluation and analysis, as well as information recovery. In the progression model, this history is captured in the evolution of the markup language document. The progression analyzer is intended to allow access to the information through the history workflow action.

Figure 5. A transaction near the end of a progression.

For example, the worker named Dave may want to go back to the previous scene to make a change to his user actions. He can select the previous scene workflow action, which processes the markup language for the previous scene. Figure 6 shows the selection of the previous scene workflow action.

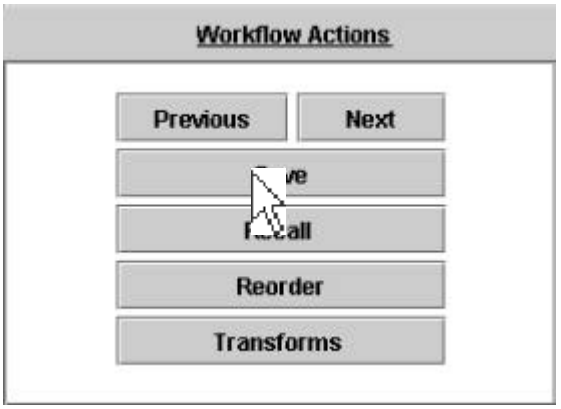


Figure 6. The workflow actions panel, which is displaying the available workflow actions.

Persistence refers to the storing and accessing of the captured history information. This research identifies that persistence and history are often combined in traditional workflow systems. [12] however, provides a separate persistence component, which gives access to the database. In the progression model, the information is saved in the mark-up document. There are also workflow actions that allow partially completed progressions and their transactions to be saved, and then recalled at a later time.

For example, the worker named Mark might want to review what he did in a previous progression. He would have previously selected the "save" workflow action to save the partially completed progression. Then when he wants to re-open the progression he selects the "recall" workflow action and the progression is displayed at the point where he saved it. He can traverse through the progression to replay his actions using the next scene and previous scene workflow actions. Figure 7 shows the file chooser for the recall workflow action.

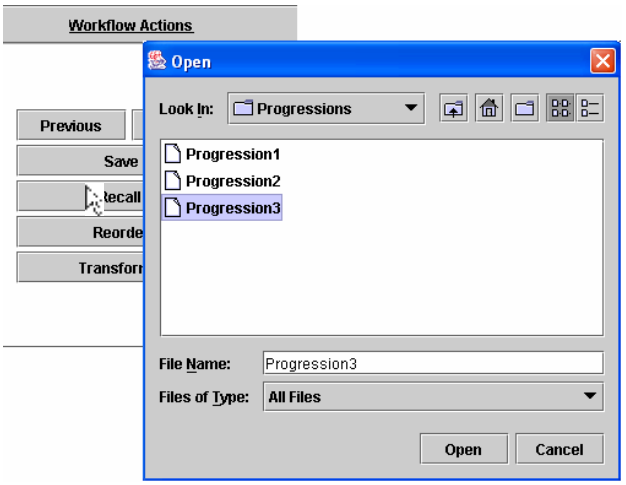


Figure 7. File chooser prompt for the recall button.

Manual intervention refers to allowing users or developers to change the organization of activities during the execution of the workflow. In the progression model, a "reorder" workflow action is provided. It allows the user to perform the workflow scenes in varying orders. This reordering action is limited to the constraints of the system.

For example, the worker named Tara may decide that she does not have the required information to complete scene two – enter credit card information, but she does have her personal information, which is required for scene three. Therefore, she would like to rearrange the scenes so she can complete as much information as possible. She would select the "reorder" workflow action to perform the third scene before the second. Figure 8 shows the new workflow panel resulting from Tara selecting the "reorder" workflow action and switching the scenes.

Figure 8. The rearranged workflow scenes.

Worklist refers to the table provided to help manage the flow of work amongst human workers concerning assigning responsibilities. In the progression model, the workflow section indicates the tasks to be completed as grouped into scenes with the corresponding worker assigned to the task. Also, some circumstances require the system to be a worker and complete part of the workflow. Therefore the interaction with the application is partially captured as well.

For example, the worker named Jim may want to determine if another worker has completed their part of the progression. He can look at the workflow panel and see which worker is

responsible for a particular scene. He can also see what he is responsible for, such as in this scenario, where he is required to provide the account information for this transaction in the fourth scene. Figure 9 shows the worklist of workers assigned to scenes in the workflow panel.

Federated Workflow addresses the issue of how workflow systems interoperate. We have not investigated the implications of the progression model and Federated Workflow. Expressing workflow with a markup language may be conducive to integrating workflow systems since the workflow is stated explicitly.

Workflow

Scene	Name	Worker	Status
1	EnterName	Dave	Complete
2	EnterAddre...	Dave	In Progress
3	EnterCredit	Tara	Inactive
4	EnterAccou...	Jim	Inactive
5	Confirm	Jen	Inactive

Figure 9. The worklist in the workflow panel

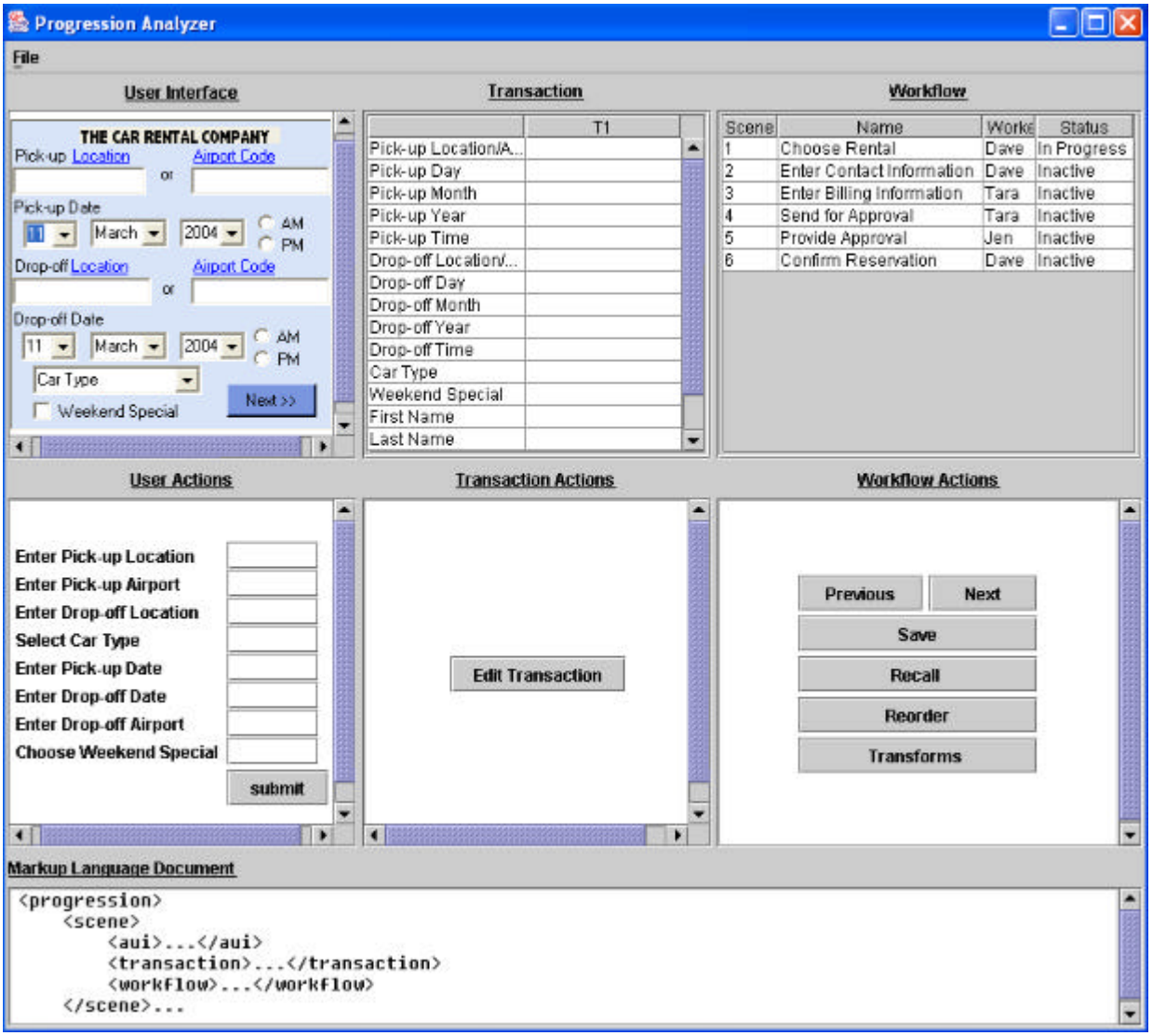


Figure 10. A screenshot of a more complex user interface and progression model in the Progression Analyzer.

CONCLUSION

User interface description languages have been designed to specify various aspects of a user interface, such as the tasks to be accomplished by the user, the abstract and concrete elements in the user interface and the user interface dialog. Workflow aspects, such as coordinating and managing tasks, are not modeled by current UIDLs. The progression model is an attempt to identify workflow issues

and to integrate workflow into a user interface description language.

We have developed a prototype system, the progression analyzer, to help refine the progression model and to investigate workflow in a UIDL. Explicitly recording and manipulating progressions allows us to dynamically change the workflow of an interactive system. Benefits include, improving the plasticity of an interactive system (workflow plasticity), providing a coarser integration with an application (transaction-based integration), and additional workflow functionality.

The progression analyzer has provided us a mechanism for studying progressions in detail. From this experience, we plan to formalize our XML-compliant language to show how workflow concepts can be integrated into a user interface markup language.

REFERENCES

1. Abrams, M., Phanouriou, C., Batongbacal, A.L., Williams, S., and Shuster, J. UIML: An Appliance-Independent XML User Interface Language. In A. Mendelson, editor, *Proceedings of 8th International World-Wide Web Conference WWW's* (*Toronto, May 11-14, 1999*), Amsterdam, 1999. Elsevier Science Publishers.
2. Annett, J., Duncan, K.D., Stammers, R.B., & Gray, M.J. (1971). Task analysis. London: Her Majesty's Stationery Office.
3. Arsanjani, A., Chamberlain, D., and et al. (WSXL) web services experience language version, 2002.
4. Azevedo, P., Merrick, R., and Roberts, D. OVID to AUIML – user-oriented interface modeling. In N. Nunes, editor, *Proceedings of 1st International Workshop "Towards a UML Profile for Interactive Systems Development" TUPIS'00* (York, October 23, 2000)., York, 2000.
5. Bernstein, A. How Can Cooperative Work Tools Support Dynamic Group Processes? Bridging the Specificity Frontier. (CSCW'00), 2000, pp. 279-288.
6. Card, S. K., Moran, T. P., and Newell, A., The Psychology of Human-Computer Interaction , Lawrence Erlbaum, 1983.
7. Haake, J., Wang, W. Flexible Support for Business Processes: Extending Cooperative Hypermedia with Process Support. (GROUP'97), 1997, pp. 341-350.
8. Hartson, H. R, Siochi, A. C., Hix, D. The UAN: a user-oriented representation for direct manipulation interface designs. ACM Transactions on Information
9. Heinl, P., Horn, S., Jablonski, S., Neeb, J., Stein, K., Teschke, M. A Comprehensive Approach to Flexibility in Workflow Management Systems. (WACC'99), 1999, pp. 79-88.
10. Luyten, K., Clerckx, T., Coninx, K., Vanderdonckt, J. Derivation of a Dialog Model from a Task Model by Activity Chain Extraction. (DSV-IS'2003), Funchal, Madeira Island (Portugal), 2003, ©Springer-Verlag 2003.
11. Luyten, K., Vandervelpen, C., and Coninx, K. Adaptable user interfaces in component based development for embedded systems. In *Proceedings of the 9th Int. Workshop on Design, Specification, and Verification of Interactive Systems DSV-IS'2002*, (Rostock, June 12-14, 2002). Springer Verlag, 2002.
12. Manolescu, D. An Extensible Workflow Architecture with Objects and Patterns. Chapter 4 in *Technology of Object-Oriented Languages, Systems, and Architectures* Theo D'Hondt, editor. Kluwer Academic Publishers, 2003.
13. Narendra, N. C., Adaptive Workflow Management – An Integrated Approach and System Architecture. (SAC'00), 2000, pp. 858-865.
14. Paternò, F., Mancini, C., Meniconi, S. ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models. (Proceedings Interact'97), Chapman&Hall, 1997, pp.362-369.
15. Paternò , F. and Santoro, C. One model, many interfaces. In Ch Kolski and J. Vanderdonckt (Eds.), editors, *Proceedings of the 4h International Conference on Computer-Aided Design of User Interfaces CADUI'2002* (Valenciennes, 15-17 May 2002), pages 143-154, Dordrecht, 2002. Kluwer Academic Publishers.
16. Puerta, A. and Eisenstein. XIML: A common representation for interaction data. In *Proc. Of the 7th International Conference on Intelligent User Interfaces* (Santa Fe, United States, January 2002), pages 69-76., New York, 2002. ACM Press.
17. Souchon, N. and Vanderdonckt, J. A Review of XML-Compliant User Interface Description Languages. (DSV-IS'2003), Funchal, Madeira Island (Portugal), 2003, ©Springer-Verlag 2003.
18. Stavness, N. and Schneider, K. A. Supporting Flexible Business Processes with a Progression Model, (IUI-CADUI 2004) Workshop: Making Model-based UI Design Practical: Usable and Open Methods and Tools , Island of Madeira, Portugal, January 2004
19. Traetteberg, H.: Modeling work: Workflow and Task modeling. In: Vanderdonckt, J., Puerta, A.R. (eds.): Proc. of 3 rd Int. Conf. on Computer-Aided Design of User Interfaces CADUI'99 (Louvain-la-Neuve, 21-23 October 1999). Kluwer Academics, Dordrecht (1999) 275–280.
20. WfMC. Workflow Management Coalition Terminology & Glossary, WFMC-TC-1011, Document Status- Issue 2.0, June 1996. Specifying Task Models. (Proceedings Interact'97), Chapman&Hall, 1997, pp.362-369

Evaluation of High-Level User Interface Description Languages for Use on Mobile and Embedded Devices

Jan Van den Bergh **Kris Luyten** **Karin Coninx**

Expertise Centre for Digital Media
Limburgs Universitair Centrum
Wetenschapspark 2
B-3590 Diepenbeek-Belgium
{Jan.VandenBergh, Kris.Luyten, Karin.Coninx}@luc.ac.be

ABSTRACT

Model-based design and the use of high-level user interface descriptions languages (HLUID) have been proposed for the design of multi-platform user interfaces. In this paper we present an analysis of required properties for HLUID so that they can be effectively used for the design of multi-platform user interfaces that can be used on mobile and embedded devices. Two HLUID, SEESCOA[1] XML and XForms basic profile, are evaluated. The former is used in a model-based design method, Dygimes, and the latter is a candidate recommendation of the World Wide Web Consortium. Based on this analysis, adaptations to SEESCOA XML and an adapted structure for the use in a new version of Dygimes, supporting the design of context sensitive user interfaces, are presented.

INTRODUCTION

The increase in diversity of computing platforms used in the every day life has caused a search for a methodology and models that can ease development for multiple platforms. One proposed method is the use of model-based design, which allows the design of user interfaces for multiple platforms and/or multiple contexts of use through the use of high-level models. These models all address one particular aspect of the design of the user interface and the application for which it is designed.

In parallel to this effort, the need for standards is recognized in industry. In the specification of user interfaces, this can be noticed in the growing compliance of current web browsers with the recommendations of the World Wide Web Consortium (W3C). This organization is actively encouraging web-authors to also separate the different aspects of the user interface through the design of new standards and new versions of existing standards. These standards also promote the separation of the different aspects of web pages: content and structure (XHTML, XForms, VoiceXML), and presentation (CSS, XSLT).

In this paper we elaborate on our work to merge certain aspects of the two approaches (model-based and standards-based declarative design) and on how well they are fit for mobile devices using a Java-based implementation. We will address into more detail the use of XForms as a high-level user interface description language and look at the effects of its use in the Dygimes design-process and tools[4]. In the discussion we will concentrate on the features offered in the basic profile of XForms since it is a version especially targeted towards mobile devices.

After the presentation of some related work we will start this paper by giving an overview of the Dygimes approach and discussing requirements for a HLUID to be used on mobile devices, followed by a detailed discussion of SEESCOA XML, the high-level user interface specification language used in the approach. After that we will shortly present the relevant aspects of XForms. The next sections give an evaluation of both approaches and the impact of a changed SEESCOA XML in an adapted Dygimes approach that supports the design of context-sensitive user interfaces The final section presents the conclusion and future work.

RELATED WORK

Several approaches have been taken to describe functionality for display on mobile devices. Nichols et al. [11] designed an XML-based description specifically targeting the use of mobile devices as a remote control for complex appliances, such as a hi-fi installation. Their description consists of the description of three different components: states, commands and descriptions. Relations between components can be expressed by groups and conditional statements. How the specification is translated in user interface objects (such as buttons and

[1]SEESCOA stands for "Software Engineering for Embedded Systems using a Component-Oriented Approach".`http://www.cs.kuleven.ac.be/cwis/research/distrinet/projects/SEESCOA/`

text fields) is left to the rendering engine. This notation is however too specific for our purposes; we want a more general approach.

A different approach is taken by Marucci et al.[9], they propose the use of model-based approach for the design of user interfaces for multiple platforms, among which a PDA or a mobile phone. In this approach, one starts by defining a global task model, which is refined for the different platforms. The refined task models can be translated to abstract user interface descriptions that include information about the composition of a dialog, in which the abstract interaction objects are split in several meaningful categories, as well as the dialog transitions. This abstract user interface description is semi-automatically translated to a concrete user interface description that is rendered at runtime. They mention support for adaptivity by the derivation of a user model, which can be updated at runtime, from the task model. The user model, then could have influence on the user interface. No detailed explanation of this adaptive process at runtime is provided. This approach differs from ours in that they use transformations between task model, high-level user interface specification and concrete user interface specification at design time. In the process of the transformations, however, some information that is useful (such as the actions of the user, the application objects manipulated through the user interface) and other platform-specific information is added.

UIML[1] is an XML-based meta-language for the specification of user interfaces that separates the different parts of the user interface description in different parts of the XML. A technical committee of OASIS[2] is is working to make it an open standard. Despite the fact that the specification forces separate specification of user interface structure, style, interaction and toolkit bindings (vocabulary), most specifications of the user interface structure still have toolkit-specific information in all parts of the user interface description. Ali and Abrams[2], however, proposed a generic vocabulary, which makes it possible to design a single user interface structure for a diversity of platforms. Although this vocabulary makes it possible to design user interfaces for more systems, it doesn't have some properties we would like it to have: it does not preserve semantic information that connects certain parts of the user interfaces. The connection between a text field and the label that describes the content of the text field are not inherently connected, while this is the case in XForms and SEESCOA XML. UIML is less suitable for small mobile devices since the display of a small user interface requires a complete vocabulary containing potentially lots of unused control specifications to be read.

XIML[8] is a meta-language that perhaps could express all information we want it to express; it allows specification of tasks, high-level and concrete user interfaces as well as a domain-specific information. Its greatest weakness and a reason we cannot be sure it meets all our needs is that it is a closed specification from one company that has a very restrictive license, which is something that is far from desirable for a specification that should be used on a multitude of devices.

Mitrović and Mena[10] proposed to use another declarative language with an open specification, XUL[6] for the description of user interfaces for mobile systems. They use XUL as the basic specification but it is transformed to HTML and WML using XSL transformations for display on mobile devices. XUL does not meet our criterion that is has to be a high-level user interface description language.

DYGIMES

Dygimes[4] is a framework that uses a model-based approach to design user interfaces for mobile devices and embedded systems. In the approach task models and high-level user interface descriptions are combined into a single specification that can be rendered by a runtime environment. Limited styling support and interaction with web services is provided. Figure 1 shows the different specifications that are used to define the user interfaces and their corresponding conceptual models as specified by Calvary et al.[3] in the reference framework for plasticity.

The central specification is a task model in ConcurTaskTrees notation[12] (CTT). It is a hierarchical task model that has four types of tasks: user tasks (tasks performed by the user without interaction with the device), interaction tasks (tasks involving an interaction of the user with the application), application tasks (performed by the application) and abstract tasks (split into two or more subtasks of different types). Tasks are interconnected with temporal operators that can be used to determine which tasks are active during the completion of a task.

The leaf-nodes of the CTT are annotated with high-level user interface descriptions, using SEESCOA XML, discussed in more detail in the following section. This enables us to generate user interfaces from the annotated task model. Custom mappings between the high-level interactors in SEESCOA XML and the concrete interactors used to show the user interface[7] can be defined. Limited styling support is also available.

We are extending the Dygimes approach to support context-sensitive user interfaces on mobile devices. For this approach we are developing a new XML-based notation that integrates information about context, tasks, and high-level user interface controls. The requirements we determined for the HLUID are:

high-level specification The user interface specification should describe the parts of the user interface at

[2]Organization for the Advancement of Structured Information Standards

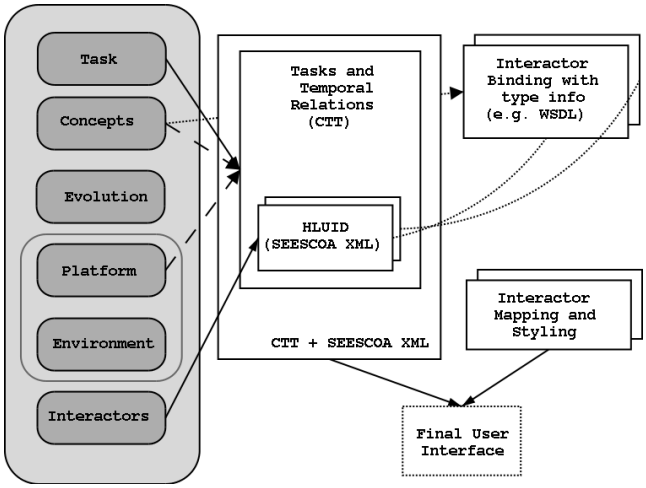


Figure 1. Dygimes specifications

a high level as SEESCOA XML does.

expressive The user interface description should not limit the possibilities of a mobile device to render the user interface. Special controls for special needs (e.g. password and date entry) may not be excluded from use due to limitations in the expressiveness of the HLUID. This also implies that (limited) extensions should be possible without the introduction of new controls in the HLUID.

customizable Styling and custom mappings should be possible and dynamically determined; designers do not want to be limited in their creativity by defaults and organizations want to be recognized.

embeddable The specification will be used embedded in an XML notation of the Contextual ConcurTaskTrees, an extension of the CTT introducing context interaction in the task model.

compact The specification should have a compact form for efficient use on mobile devices and embedded systems.

local and remote interaction It should be possible to handle both local and remote interaction involving both sending and receiving data.

SEESCOA XML

SEESCOA XML is a XML-based language for the specification of high-level user interface descriptions. It was a stand-alone specification that is now used in conjunction with the ConcurTaskTrees notation [12] in the Dygimes process[4].

Characteristics

The most important characteristics of SEESCOA XML are:

Multiple platform support The use of high-level descriptions of the controls as well as the ability to make logical groups and the specification of possible splitting of groups should make rendering on multiple platforms possible.

Local and remote interaction The specification allows the definition of different communication channels between the user interface and the program logic. Currently XML-RPC, SOAP and local java event handling are supported.

Human readable The specification should be readable by both man and machine.

Separation of concerns The user interface specification should be as independent as possible of the presentation and the programming logic.

Scalable for embedded systems The specification is especially targeted towards embedded systems, e.g. suitable to be stored in limited memory space like on a Radio Frequency Identifier Tag,...

Specification

A typical user interface description using SEESCOA XML is organized as shown in figure 2. The group tag identifies elements of the user interface that have some kind of logical relation: e.g. all the controls to specify a date. Groups can contain other groups, controls and constraints. The constraints included in a group specify the spatial relation between the different children of the group.

There is a limited set of controls that can be presented. Each control is specified within a *interactor*-tag that contains a control-specific tag. All controls have an *info* tag that can be used to describe the purpose of the control. The information in the info tag can be displayed as a separate control or as a part of the main control as can be seen in figure 2, showing a partial user interface description for making an appointment.

SEESCOA XML provides no information about the presentation of the user interface, except for layout purposes. One can specify constraints for controls and groups within a common group. Four linear spatial constraints are supported (left, right, above and under) and are honoured as much as possible. When it is impossible to place all controls on the screen as desired, the user interface can be split into several layers. The designer preferences are taken into account here: groups can also be marked "non-splittable".

There is limited styling support for SEESCOA XML. Styling is specified using an XML format and supports mapping of the controls specified in SEESCOA XML to the controls that are rendered as well as styling of the mapped controls. Customized mappings can be created for all controls of a specific type, for a subset thereof or for a single control. The selection of the appropriate mapping-rule is determined by the type of control and by (part of) the name of the controls[7]. The appearance of the controls can be customized per (set of)

```
<?xml version="1.0"?>
<ui>
<title>Make Appointment</title>
<group>
    <group name="exampleGroup">
        <interactor>
            <range name="hour">
            <info>hour</info>
            <min>0</min>
            <max>23</max>
            <start>9</start>
            <tick>1</tick>
            </range>
        </interactor>...
        <interactor>
            <textfield name="description">
            <info>Description</info>
            <size>40</size>
            </textfield>
        </interactor>...
        <interactor>
        <button name="ok">
            <info>OK</info>
            <action type="Java">
               <!-- action description-->
            </action>
        </button>
        <constraints>
            <!-- spatial contraints-->
        </constraints>
    </group>
</group>
</ui>
```

Figure 2. Example SEESCOA XML

mapping rule(s). The implementation of styling, however, is currently only specified for platforms supporting Java AWT.

The binding with the functional core is provided through *action* tags, which have a *type*-attribute that specifies the invocation protocol. Action elements are generic elements that can be used as sub elements for all available controls. Each manipulation of the control will execute the invocations specified within the action elements. The XML within the action element is not restricted within the XML Schema; this allows developers to attach their own invocation protocols to controls. At the time of writing we have successfully used direct method invocation (dmi), XML-RPC and SOAP for this purpose. Within the SEESCOA system it was possible to invoke the specific SEESCOA components.

XFORMS

XForms[5] is a W3C recommendation for the design of forms for web-based applications using an XML-based specification. It is designed to be the replace the current form-specification in HTML in the modularized XHTML 2.0 specification. Its possibilities, however, are much richer than those offered by HTML form's.

Characteristics

The characteristics of XForms can be specified as:

XML for instance data This makes direct validation and processing by the application backend possible (no need for marshaling of data). It also ensures that data is internationalization ready.

Multiple platform support High-level description of user interface controls makes multiple device support possible.

XML event handlers The use of declarative event handlers for the most common events reduces the need for imperative scripts for this purpose.

Strong typing All instance data is strongly typed enabling client-side checking of supplied data and saving a round-trip for invalid data.

Extensibility All parts of XForms are extensible through the use of namespaced attributes and/or tags.

Reuse Existing schemas are reused (e.g. XML Schema for typing and XML Events for event handler specification)

Enhanced accessibility XForms separates content and presentation. User interface controls encapsulate all relevant metadata such as labels, thereby enhancing accessibility of the application when using different modalities.

There are different conformance levels for the XForms standard; full and basic profile. The basic profile is meant for enabling XForms parsers in mobile devices. Since this is what we are interested in, we will only discuss the XForms basic profile from now on.

One of the special properties of XForms is that it is not meant to be used on its own, but rather that is would be embedded in other specifications, such as XHTML. It therefore has no root-tag, as other XML-based languages have and can be used in different places in a XML document. Differentiation between XForms and the data in the host language is done by the use of namespaces.

Specification

A XForms specification consists of two major parts, which are embedded in a host-language as depicted in figure 3:

model this part of the description describes the types of information that will be manipulated or used by the form controls and the submission method that is used to communicate with the application logic

form controls a high-level description of the form controls that are used to input, manipulate and/or output data or submit data to a server

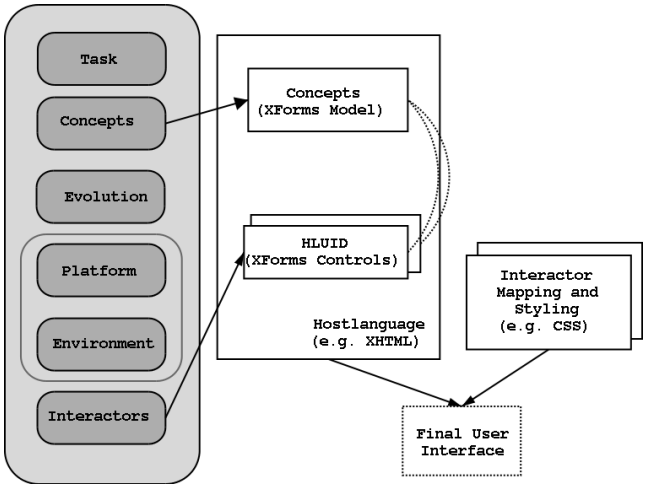


Figure 3. XForms specifications

```
<xforms:model>
    <xforms:instance>
        <appointment ...>...
             <hour>9</hour>
             <description/>
        </appointment>
    </xforms:instance>
    <xforms:bind constraint=".&lt;24"
      nodeset="/appointment/hour"
      type="xsi:nonNegativeInteger"/>
    <xforms:submission
      action="examples.Appointment"
      method="dygimes:dmi" id="submit"/>
</xforms:model>
```

Figure 4: XForms model example, corresponding controls in figure 5

We will discuss the most relevant parts the specification into more detail.

Model

The model part consists of two parts; the first describes the structure of the *instance* data that will be submitted by a *submission* specified in the second part. The second part is also used for the specification of the type of the elements in the instance data.

Datatyping of the instance data specified in the model can be done by binding the instance data to simple XML Schema datatypes[3] and XForms data types using the bind tag and XPath. Further restriction on the possible values of instance data can be posed by specifying a *constraint*, an XPath expression evaluating to *true()* or *false()*. The *xforms:bind* tags in figure 4 show examples of such expressions. The value of hour is a non-negative number and constrained to be smaller than 24.

[3]in the Full Profile almost all XML-schema datatypes can be used

```
<xforms:group>
    <xforms:input ref="/appointment/hour">
        <xforms:label>Start time</xforms:label>
    </xforms:input> ...
    <xforms:textarea
       ref="/appointment/description">
        <xforms:label>Description</xforms:label>
    </xforms:textarea>
    <xforms:submit submission="submit">
        <xforms:label>OK</xforms:label>
    </xforms:submit>
</xforms:group>
```

Figure 5. XForms controls example

The *xforms:submission* tag specifies how the information in the *xforms:instance* should be handled. The method describes the type of submission, which can be arbitrary, although a limited set of guaranteed methods are described in the recommendation, such as the different HTTP-methods and (local) storage.

User Interface

XForms supports similar controls as SEESCOA XML as can be seen in figure 8. In most cases controls in both languages can easily be matched. There are a couple of exceptions: SEESCOA XML has no special control for password entry, nor for multi-line text input and file upload. A special remark should be made about the controls for information display and for giving commands.

The *output* control has no specific meaning other than that it is used to display data described in the model. How the data is presented to the user is only specified in the mapping/styling. This contrasts to the approach in SEESCOA XML where different controls are used for the display of text or visual data (images).

The command functionality is represented in SEESCOA XML by a single control, the *button*. XForms, however, has two separate controls for this purpose; *trigger*, which can be used to perform some manipulation of the user interface, and *submit*, which is used to submit data to the functional core.

XForms also supports dynamic user interface specifications through the use of *switch*, *repeat* elements which allow dynamic insertion and removal of controls of the user interface without the need for a (separate) script. This makes it possible to specify dynamic lists in XForms, which can be difficult or impossible in other user interface description languages (such as SEESCOA XML).

Figure 5 shows the controls that can be used in conjunction with the model in figure 4 to generate an interface equivalent to the one specified in figure 2. One difference that can be noted is that instead of an *input* is used instead of a *range* which is used in the example with SEESCOA XML. This is caused by the restrictions

control modifier	SEESCOA XML	XForms
information about ...	info	label
extra information	*not supported*	hint
help	*not supported*	help
error notification	*not supported*	alert
action	action	action
selection items	item	item
selection item group	*not supported*	choices
size hints	*not supported*	appearance

Figure 6: Tags providing actions or information about control

on type binding for the range in XForms, which cannot be bound to an integer-based type. Both the specification in SEESCOA XML and the one using XForms support the same restriction on the input for the hours and minutes.

Styling can be done by using CSS, although to reach all the desired styling support (including mapping to concrete controls) future standards are needed. Furthermore, XForms leaves the specification of the attributes needed for style up to the containing document structure.

EVALUATION HLUID

We will now revisit our requirements for a HLUID that appropriately supports mobile and embedded systems. The first requirement we posed, high-level specification, is met by both SEESCOA XML and XForms. Both have a similar level of abstraction as can be seen in figures 8 and 6. The level of abstraction is also similar to that used in other approaches[2, 9].

To evaluate expressiveness regarding mobile devices, we looked at the MIDP 2.0[13] (Mobile Information Device Profile). This specification for java virtual machines on small mobile devices should give a good idea on what functionality will be reliably present on mobile devices and, thus, should be supported. Figure 8 shows that the expressiveness of XForms meets, and even exceeds our needs (there is no standardized way for giving hints and help about certain controls in MIDP 2.0). The typing and constraint support of XForms allow a MIDP 2.0 rendering engine to use a DateField a a Textfield with constraints (see figure 7), and more generally enable native support of type-specific controls. XForms also supports alerts, notifications given on wrong data-entry (figure 6). Alerts are also present in MIDP 2.0 (class `Alert`. These things are not possible with the current SEESCOA XML and thus SEESCOA XML, as is, does not meet the criterion of expressiveness. The XForms specification on the other hand does not provide layout-hints, nor provisions for a canvas that can be used for image display, etc.

Both HLUID allow customization and thus both satisfy the third requirement. Both specifications are also compact by design and thus satisfy the fourth requirement.

Type	MIDP 2.0 Controls	XForms
text	TextF.(ANY)	string
numeric	TextF.(NUMERIC)	integer
URL	TextF.(URL)	anyURI
email	TextF.(EMAILADDR)	*xpath*
phone number	TextF.(PHONENUMBER)	*xpath*
decimal	TextF.(DECIMAL)	decimal
date	DateF.(DATE)	date
date/time	DateF.(DATE_TIME)	dateTime
time	DateF.(TIME)	time

Figure 7: Typing contraints for Textfield and Datefield in MIDP 2.0 and corresponding XForms typing

The presentation of the controls can be based on the type of data in XForms and can also be customized using style sheets. The support in current software for advanced styling is limited but recommendations in development promise standardized support through style sheets. Support for data-specific controls is not possible using the original SEESCOA XML, however the controls specified in XML can have designer-specified instantiations.

The last requirement is that of the possibility of both local and remote interaction. This is a drawback of the XForms 1.0 specification; it is mainly targeted at web applications and thus there is no specification for local interaction, furthermore the interaction with the application logic is limited to submission of data.

In order to meet all our requirements and minimize the implementation required for rendering on e.g. a MIDP 2.0 device, we will make the controls of SEESCOA XML type-aware. A set of minimal supported data-types will be provided, data types as well as the "concepts" are described in a separate section, the earlier given example in the new SEESCOA XML is shown in figure 9. The full specification of the actions will also reference to the data in this section rather than the controls showing the data (not shown in the figure).

INTEGRATION IN TASK-BASED NOTATION

The previous sections discussed the use of XForms and SEESCOA XML as a high-level user interface description language, without considering the specification in which it will be incorporated. This section will discuss the use of HLUID in the task model.

In the discussion of Dygimes, we mentioned that tasks are annotated wit HLUID. This annotation ensures that navigation and composition can be derived from the Enabled Task Sets (ETS), which are defined as "a set of tasks that are logically enabled to start their performance during the same period of time" in [12]. Fig. 10 shows a preliminary version of the annotation tool.

Integration of the adapted SEESCOA XML into the

control	SEESCOA XML	XForms	MIDP 2.0
text input	textfield	input	TextField/TextBox
multiline textfield	textfield	textarea	TextField/TextBox
single selection	choice (choicetype=”single”)	select1	ChoiceGroup/List
multiple selection	choice (choicetype=”multiple”)	select	ChoiceGroup/List
password field	*not supported*	secret	TextField (PASSWORD)
information display	label, canvas	output	StringItem, Canvas
range	range	range	Gauge (int)
upload file	*not supported*	upload	*not supported*
command	button	trigger, submit	Command
group	group	group	Form

Figure 8. Supported controls by SEESCOA XML, XForms and MIDP

```
<ui>
<title>Make Appointment</title>
<group>
    <group name="exampleGroup">
        <interactor>
            <range name="hour"
       ref="/application/hour"
               model="app">
            <info>hour</info>
            <min>0</min>
            <max>23</max>
            <tick>1</tick>
            </range>
        </interactor>...
        <interactor>
            <textfield name="description"
       ref="/application/description"
       model="app" type="multiline">
            <info>Description</info>
            </textfield>
        </interactor>...
        <interactor>
        <button name="ok">
            <info>OK</info>
            <action type="dmi">
               <!-- action description -->
            </action>
        </button>
        <constraints>
            <!-- spatial contraints>
        </constraints>
    </group>
</group>
</ui>
```

Figure 9. Example new SEESCOA XML

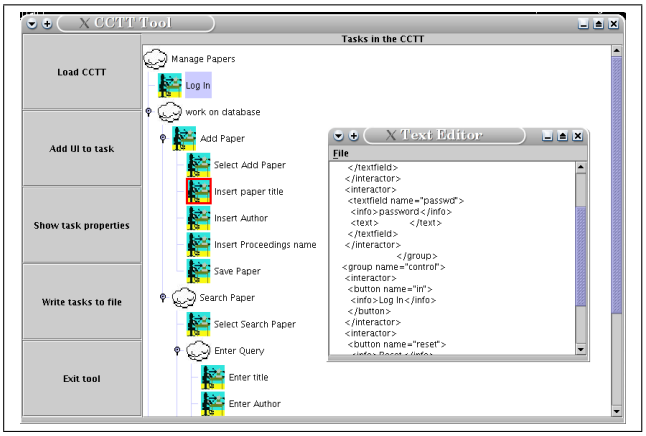


Figure 10. The CTT annotation tool

CTT will create a partial duplication of information since both specification provide their own method of specifying data relevant to respectively a HLUID and a task. This, together with the need for specification of context within the model to enable creation of context-sensitive user interface, led to the creation a separate specification, Dygimes XML. The specification will consist of three parts: specification of *concepts*, specification of *context* and specification of tasks as can be seen in figure 11. Tasks in Dygimes XML can have HLUID attached to them. The HLUID descriptions are specified using the adapted SEESCOA XML with references to the *concepts* for data-binding and type information. Both the concepts and the context will be specified using the XForms model.

CONCLUSION

We have discussed high-level user interface descriptions on mobile devices using two examples, SEESCOA XML and XForms, and their application to mobile devices. We did this by identifying requirements and evaluating to which extend these specifications satisfied those requirements. We found that the existing SEESCOA XML was reasonably suited for the specification of user interfaces for mobile systems. However, the lack of knowledge about the type of the manipulated data and

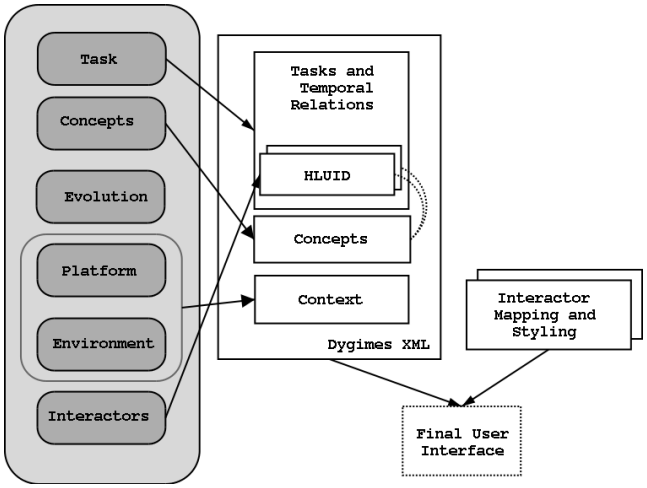


Figure 11. CCTT specifications

the direct link with widgets to extract data were noted as required features for the generation of flexible interfaces from a HLUID.

We specified adaptations to SEESCOA XML, necessary to fulfill all requirements. The new specification of SEESCOA XML will be made available as part of the Dygimes XML specification, which will integrate the XForms data model, at the time of the workshop. We have started implementation of a MIDP 2.0 renderer for the (new) SEESCOA XML and support for the full Dygimes XML is planned.

ACKNOWLEDGEMENTS

Our research is partly funded by the Flemish government and European Fund for Regional Development. The SEESCOA project IWT 980374 (1999-2003) is directly funded by the IWT (Flemish subsidy organization).

REFERENCES

1. Marc Abrams, Constantinos Phanouriou, Alan L. Batongbacal, Stephen M. Williams, and Jonathan E. Shuster. UIML: An appliance-independent XML user interface language. *WWW8 / Computer Networks*, 31(11-16):1695–1708, 1999.

2. Mir Farooq Ali and Marc Abrams. Simplifying construction of multi-platform user interfaces using uiml. In *Proceedings of UIML 2001*, March 8–9 2001.

3. Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Nathalie Souchon, Laurent Bouillon, and Jean Vanderdonckt. Plasticity of user interfaces: A revised reference framework. In *Task Models and Diagrams for User Interface Design*, pages 127–134, Bucharest, Romania, July 18-19 2002. TAMODIA 2002.

4. Karin Coninx, Kris Luyten, Chris Vandervelpen, Jan Van den Bergh, and Bert Creemers. Dygimes: Dynamically Generating Interfaces for Mobile Computing Devices and Embedded Systems. In *Human-Computer Interaction with Mobile Devices and Services, 5th International Symposium, Mobile HCI 2003*, pages 256–270, Udine, Italy, 8–11 2003. Springer.

5. World Wide Web Consortium. *XForms 1.0, W3C Recommendation 14 October 2003*. World Wide Web, http://www.w3.org/TR/2003/REC-xforms-20031014/.

6. danm@netscape.com. *Introduction to a XUL Document*. World Wide Web, `http://www.mozilla.org/xpfe/index.html`, october 1999.

7. Jan Van den Bergh, Kris Luyten, and Karin Coninx. A Run-time System for Context-Aware Multi-Device User Interfaces. In *HCI International 2003, Volume 2, Crete, Greece*, pages 308–312, June 2003.

8. Jacob Eisenstein, Jean Vanderdonckt, and Angel Puerta. Applying Model-Based Techniques to the Development of UIs for Mobile Computers. In *IUI 2001 International Conference on Intelligent User Interfaces*, pages 69–76, 2001.

9. Luisa Marucci, Fabio Paternò, and Carmen Santoro. *Supporting Interactions with Multiple Platforms Through User and Task Models*, pages 217–238. Wiley.

10. Nikola Mitrović and Eduardo Mena. Adaptive user interface for mobile devices. In *Interactive Systems. Design, Specification, and Verification. 9th International Workshop DSV-IS 2002, Rostock (Germany)*, pages 47–61. Springer Verlag, June.

11. Jeffrey Nichols, Brad Myers, Thomas K. Harris, Roni Rosenfeld, Stefanie Shriver, Michael Higgins, and Joseph Hughes. Requirements for automatically generating multi-modal interfaces for complex appliances. In *IEEE Fourth International Conference on Multimodal Interfaces*, pages 377–382, 2002.

12. Fabio Paternò. *Model-Based Design and Evaluation of Interactive Applications*. Springer, 2000.

13. James E. Van Peursem. *JSR 118: Mobile Information Device Profile 2.0*. World Wide Web, `http://jcp.org/en/jsr/detail?id=118`.

useML: A Human-Machine Interface Description Language

Detlef Zuehlke, Kizito Mukasa, Alexander Boedcher, Achim Reuther
Center for Human-Machine Interaction
Institute for Production Automation
Kaiserslautern University of Technology
67663 Kaiserslautern, Germany
Phone: +49 631 205 3570
Fax: +49 631 205 3705
[zuehlke|mukasa|boedcher]@mv.uni-kl.de|reuther@useml.de

ABSTRACT
This paper describes a XML based user interface description language called useML, which was developed at the Center for Human-Machine-Interaction at the Kaiserslautern University of Technology. The language allows a model based, task oriented and platform independent description of user interfaces in production environments. The main concept is the abstraction of user tasks and interactions into use objects and elementary use objects. The description forms one central document called use model, from which platform specific prototypes can be generated. useML supports the analysis, the structuring and parts of the design phase of the Useware development process, which is also introduced in this paper. The structure of useML is described and its usage is demonstrated with an example. An outline of some related languages has also been given.

Keywords
XML, MB-UID, UIDL, Useware, UIML, XIML, use model, use object, task modeling, interaction modeling

INTRODUCTION

The development of user interfaces for machines differs from the classic software development in a number of ways. First, the development is done offline (in most cases on a PC), but the application will run on the machine. The development PC is always equipped with input devices like mouse and keyboard and has a large display. On the other hand, there are no such input devices on the machine and the display is normally small (in most cases less than 12"). Second, most developers have less experience with operating the machine. Therefore the knowledge gap between developers and machine users is in most cases very large. Moreover, developers and machine users have different thinking philosophy. While developers think in terms of software, machine users think in terms of hardware and tasks. Third, the operation on a machine is different from that on a PC. While the office user "pushes" or "drives" his process, the machine user is being "pulled" or "driven" by the machine. Any wrong input can result into irrevocable and expensive damages to the machine, the product and in worst case to the user. Thus careful design of Useware[1] to support user tasks is required. This can be achieved by following a systematic development approach that allows considering users, their tasks and their experiences. Additionally, a user interface description language (UIDL) should support developers towards the accomplishment of these goals. For example, it can provide ways of defining task flow and relationships between tasks as well as defining constraints related to the task execution, e.g., execution place, access rights, etc. A combination of development process and UIDL is therefore required. In the first part of this paper, a process for developing machine Useware will be introduced. The second part describes details of a new UIDL named useML. An example of applying useML is also available. The third part presents a briefing of related work and finally is a conclusion and future work.

THE USEWARE DEVELOPMENT PROCESS

Systematic development supports user and task consideration as well as helping the developer manage the complexity of today's UIs. Following this approach, the development process includes four main phases namely analysis, structuring, design and realization. These phases are not strictly isolated but overlapping. Also a continuous and iterative evaluation connects them as Figure 1 shows.

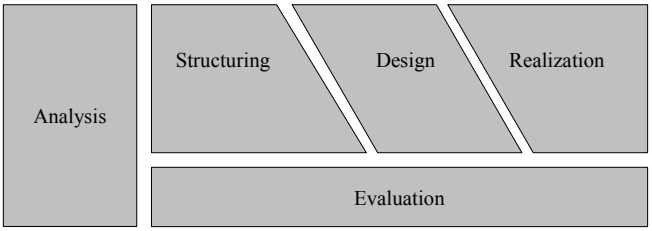


Figure 1: The Useware development process

Starting with the **analysis**, information about the users, their tasks, their working environment, their knowledge

[1] Useware is a collection of hardware and software that is required for human-machine interaction.

and their mental models is collected. Also information about available devices as well as machines and their functions is documented. An analysis of this information results into user groups and tasks. The main goal of the **structuring** phase is then to properly organize user tasks and to link them with machine functions and data that is required for the execution. This also includes attaching restrictions to tasks like for example, allowed user groups and possible execution machines and locations (local, regional or global). For instance some tasks or task options may be available on a machine with a big display but not on PDAs, or remote access to some tasks may be forbidden due to security reasons. The result of this phase is a hardware independent task structure enriched with the extra information mentioned above. By tailoring this structure to a specific hardware, a task-oriented navigation structure is obtained. This is where the **design** phase starts. The main task here is to make the user interface visible and accessible to the user on a specific visualization device. This is to say that hardware constraints like display size must now be taken into consideration. By applying special ergonomic design rules for machine user interfaces, the operation panel is divided into different areas, which will contain logically identical widgets. For example, there may be an area for navigation buttons, for function keys and for data display [12]. The data display area differs from the other two while it contains dynamic content. It is a main area where the user can view, enter or change data. The Display area can further be partitioned into message and status areas and an area for data input and output (see Figure 2).

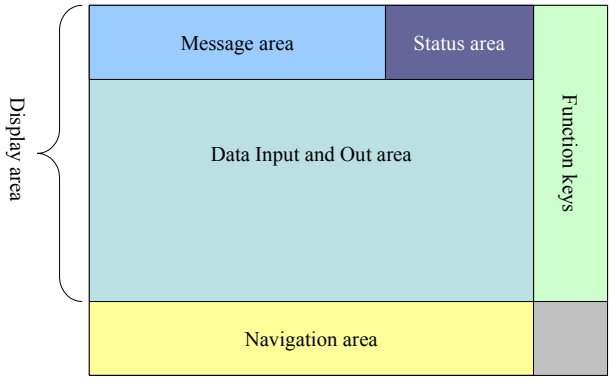


Figure 2: An example of layout for a machine user interface

Diagrammatic prototypes like wire-frame mockups or abstract layout diagrams can be used in this phase [2]. After the design phase, the user interface can now be realized by programming.

TADEUS suggests almost similar steps for the development of a user interface that reflects the world of the tasks as perceived by users and that takes into account individual skills and preferences, as far as they are required and relevant for task accomplishment and for interaction. The steps included are the Work analysis that results into the business intelligent model, the Task-based design that results into an application model and the Workflow-based prototyping that results into a running user interface [8].

The Useware development process presented here includes a continuous evaluation that must be done with users, hence ensuring their full participation in the development process. This evaluation should therefore be done throughout the Useware development and not once at the end of the development process, since errors can occur at any stage. If these are detected in early stages, correcting them is easier and not as expensive as in advanced stages.

It is obvious that proper implementation of such a development process and the integration of ergonomic design rules requires tool support accompanied with a description language. In order to demonstrate the possibility of integrating ergonomic rules in the development process, a tool called autoCAID was developed [14, 15]. autoCAID does not only integrate the rules, it also guides the developer throughout the whole development process, giving detailed and context dependent help. The lack of a proper description language hinders data exchange between autoCAID and other applications. A UIDL called useML has recently been defined at the center for human-machine interaction as a solution to this problem.

THE USEWARE MARKUP LANGUAGE (useML)

Elements of useML

useML is a XML derivative for modeling Useware [6]. The useML notation follows a usage oriented approach. The main components are the <use objects> (UOs) and <elementary use objects> (eUOs), which are contained in the root element called <use model>. While the UOs define tasks, the eUOs define actions that are parts of the tasks. Since a task can also be part of another task, a UO can also contain other UOs, hence forming an aggregation (see Figure 3). In this paper, actions will be abstracted to interactions to emphasize the bidirectional nature of the interaction between the user and the machine.

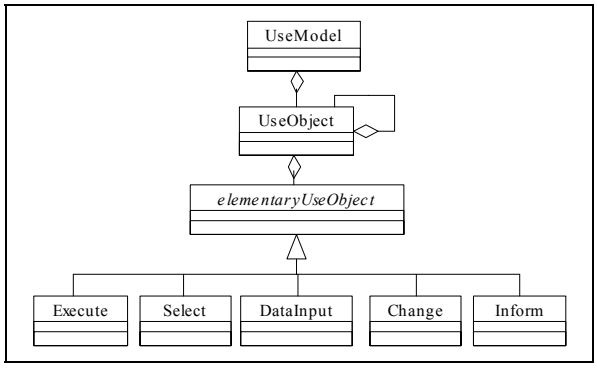


Figure 3: A class diagram of main classes (elements) of useML

The Root Element <use model>

As specified in XML, there must be one root element in file that will contain the whole content. This role is played by the element <use model> in useML as shown on Figure 4.

Besides containing other elements, a date, authors, an id or even filters for the prototypes can be defined.

```
<!ELEMENT usemodel (filter*, name,
                    date?, author*, short_description?,
                    useobject*, comments*)>
<!ATTLIST usemodel
    author CDATA #IMPLIED
    id CDATA #IMPLIED
    status CDATA #IMPLIED
>
```

Figure 4: DTD for the <use model>

The <filter> is an optional element, which can be used by style sheets for rendering prototypes on different hardware platforms. If available, this is the only place where the name of the target platform and the size of its display are mentioned. Otherwise, the <use model> remains free of any platform specifications.

The Element <use object> (UO)

Figure 5 shows a DTD for the UO as defined in the first version of useML. The eUOs have been highlighted for better identification. As mentioned above, a UO represents a task, for example "View operating statistics". This must be associated with a name and can have conditions, a short description and comments. The element <mapping> maps the task to the user groups, interaction devices, machine functions and access location and will be described later. If the attribute "sequence" is specified, it determines a sequence of steps in performing the task.

```
<!ELEMENT useobject (name, mapping*, condition*, short_description?,
                     (useobject | execute | select | datainput|
                      change | inform |)*, comments*)>
<!ATTLIST useobject
    type CDATA #IMPLIED
    id ID #IMPLIED
    sequence CDATA #IMPLIED
>
```

Figure 5: The DTD of the <use object>

The <elementary use objects> (eUOs)

useML uses eUOs to model the interactions of the user with the machine. Five elementary interaction types can be identified; execute, select, data input, change and inform.

- execute
  These are interactions where the user directly triggers a machine function. This results into direct execution of machine functions. For example, "press the start button".
- select
  By this interaction, the user can select one or many values from a set of values that already exist in the machine system. This selection can lead to changing a parameter in the machine control, for example, changing the unit of speed from km/hr to m/s, or to triggering a machine function, e.g., changing the machine operation modus from "automatic" to "manual" by selecting the required modus.
- data input
  This involves input of one absolute data value into the machine system. Although the machine can suggest a default value, it will be overwritten by user input. The machine does not know the data value (other than the default one, if available) before user input. It means that the input comes from outside the system. An example of this interaction type is "enter the user name".
- change
  This interaction is basically like the data input. The difference is that the system provides a way of manipulating input relative to an existing value. This means that the data already exists in the system. For example the user can increment the speed from 15m/s to 17m/s by using a Toggle-Wheel-Switch with 2 as an incrimination factor.
- inform
  Interactions that involve output of information from the machine system are of this type. The user queries the machine for some information, for example its status.

Any interaction of the user with the machine can be reduced to one of these elementary interaction types.

Figure 6 shows a DTD for the eUO <select>. The DTD indicates that many values can be associated with a eUO as options. The attribute "confirmation_required" determines whether the user expects confirmation from the system or not. Another attribute "user_focus" indicates the main goal of the user interaction. The goal can be "information" or "manipulation". Depending on this goal, special measures can be taken in the design phase to support the user appropriately. If the attribute "list_only" is set to "yes", then only values selected from the list are valid. Otherwise the user can enter new values that will be added for the next selection. In order to limit the number of values per selection, an attribute "multiple_selection" is used. Sometimes it is required that a machine function be executed automatically after selection. The name of this function is stored in the attribute "trigger_function".

```
>
<!ELEMENT select (name, mapping*,
                  condition*, short_description?,
                  option*, comments*)>
<!ATTLIST select
    confirmation_required (yes|no) "no" #IMPLIED
    user_focus (information | manipulation) #IMPLIED
    list_only (yes|no) #IMPLIED
    multiple_selection CDATA #IMPLIED
    trigger_function CDATA #IMPLIED
>
```

Figure 6: DTD for the eUO <select>

Other eUOs have a similar DTD with some few exceptions.

The Element <condition>

This element is used to define a condition, which can be associated with a UO or eUO. The DTD in Figure 7 indicates that this can be a pre-condition or a post-

condition. The logical connection of conditions is yet to be defined.

```
<!ELEMENT condition (#PCDATA)>
<!ATTLIST condition
        type (pre | post) #IMPLIED
>
```

Figure 7: DTD for the element <condition>

The Element <mapping>

In order to specify which user group should perform which task, at which machine or interaction device and from which location, the element <mapping> is defined. This element can also associate a task or an interaction with machine functions to be executed (see Figure 8).

```
<!ELEMENT mapping (user_group* | location* |
                   function* | machine*)*>
```

Figure 8: DTD for the element <mapping>

- The <user_group> element specifies groups that are allowed to perform the task or interaction. Mapping a user group to a task or an interaction means that only members of the specified user group are allowed to perform the task or interaction. This restriction is applied top down in the task hierarchy, i.e. from the task at the upper level to tasks and interactions at the lower level. Thus if a user group has been excluded from performing an upper level task, it will also not be allowed to perform underlying tasks and interactions. The restrictions are hardware independent. In most cases, they depend on the company organization and security reasons. Normally machine users are organized in hierarchical groups that reflect their tasks and knowledge. In the production automation area for example, the lowest group is that of operators. These are only responsible for keeping the machine running. For example, they will load or unload the machine with work pieces and then start the production process. They may also control the performance of the machine and the quality of products. Following the operators is the group of supervisors. They prepare production plans, allocate operators to machines and shifts, perform diagnostic and apply solutions, should any breakdown or malfunction occur. Of course they can also access all tasks performed by operators. The third and highest user group consists of experts. These have high technical knowledge and are mostly available during the installation and when severe malfunctions occur. They have access to all machine functions and are allowed to perform all tasks.
  In this way tasks may be made available for some user groups but hidden or deactivated for others.
- The <location> element specifies possible access areas for a task or an interaction. Possible values are "local" meaning that the operation is only allowed direct at the machine, "regional" that allows operation from another machine within the organizational intranet, and "global" the corresponds to operation from the internet.
- The optional <function> element can be used to link the task or interaction to specific machine functions necessary for its execution.
- Should need be, a task or interaction can be explicitly allowed on some machines or devices. This is specified in the <machine> element. Options for these tasks will then be made available on the specified machines only.

By using the <mapping> element, tasks and interactions can be made available or hidden for some users, at different locations and on different machines or devices.

UI Prototyping with useML

useML supports the prototyping of user interfaces especially for testing the functionality of the UI in XML based applications like web browsers and wireless devices. Figure 9 shows that by using special style sheets and script languages like Java Script, the <use model> can be transformed into dynamic web based GUI-Prototypes for different target platforms. This is very important for the communication with end users, since they can use the prototypes to evaluate the structure. For example, the users can test the accessibility of some interface functionalities before the implementation starts. It is important to note that the goal of prototyping here is to test the "functionality" and not the "visualization" of the UI.

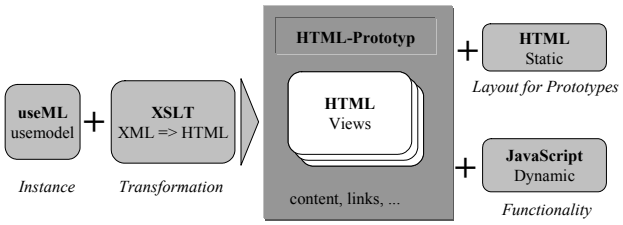


Figure 9: Rendering useML to different hardware platforms

An example of using useML

The following example should help to demonstrate UI description with useML. It is a simplified description of user tasks in a machine network, i.e. the machines are connected. This will be called "system" for simplicity.

The identified tasks are grouped into the following system contexts:

- Operation
  In this context, the machine operates normally. There are no malfunctions or any production interruptions. The user can select any machine and view its production statistics, its status, its production parameters or its configuration.
- Diagnostic
  In this context, the user can view and quit warnings, read about possible problem solutions or document his individual solutions. He can also view maintenance parameters.
- Configuration
  This is a context where users can make major setups.

- Information
  A context for getting information about the system, process and other production related information.

Organizing user tasks in contexts supports the user to have a better mental model of the machine and tasks [9].

Since considering details of all contexts might exceed the limits of this paper, only the context "Operation" will now be considered in details. Figure 10 shows user tasks and interactions as well as the machine, the location of operation and the machine function that are provoked during the interaction. Tasks consist of other tasks, interaction or both.

Task/Interaction		User Group	Machine	Location	Function
View operating statistics					
	select time interval	Supervisor	all	all	T
	get operating hours	Supervisor	all	all	hOp
	get non production time	Supervisor	all	all	stopTime
	get total output	Supervisor	all	all	Out
	get output per day	Supervisor	all	all	dOut
	get average output per hour	Supervisor	all	all	avgOut
	get number of production interruptions	Expert	all	all	iCount
	reset statistics	Expert	all	local	sReset
View plant statistics					
	select time interval	Supervisor	all	all	T
	get plant operation time	Supervisor	all	all	plantOp
	get plant non opertation time	Supervisor	all	all	stopOp
	get expected production	Supervisor	all	all	Pex
	get plant eficiency	Supervisor	all	all	Peff
select machine		all	all	all	M
change revolution speed		Supervisor	all	local	R
change production rate		Operator	all	local	Pr
get engine temperature		all	all	all	eTemp
get engine power		all	all	all	eP
get medium temperature		all	all	all	mTemp
start sample production		all	all	local	sP

Figure 10: User tasks and interactions in the context "Operation"

Taking the task "View operating statistics" as an example, we see that it has the following interactions;

1. select time interval,
2. get operating hours,
3. get non production time,
4. get total output,
5. get output per day,
6. get average output per hour,
7. get number of production interruptions and
8. reset statistic.

While the first six interactions can be performed by a supervisor, at all machines and from any location, the last two are performed by an expert. The last interaction must be performed direct at the machine.

Considering the organization of tasks and interactions as a hierarchical tree, the interactions "select machine", "change revolution speed", "change production rate", "get engine temperature", "get engine power", "get medium temperature" and "start sample production" can be considered to be at the same level as the tasks "view operating statistics" and "view plant statistics".

During the interaction "select time interval" the user can select only one time interval for the statistics he needs to view. Possible values are; "today", "last day", "last week" and "last month". As expected, this selection is modeled by the eUO <select> (see Figure 11). The name of the eUO is derived from the interaction without the word "select". Hence the eUO has the name "Time interval". The attribute "multiple_selection" that is associated with this eUO has the value "false" indicating that only one selection is allowed. The default value of this selection is "today" as indicated by its attribute "selected" which have the value "true".

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="useML_Viewer.xsl"?>
<usemodel author="Reuther (ZMMI - TU Kaiserslautern)"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="useML_schema.xsd" id="1">
    <!-- Filter options for prototypes -->
    <filter prototyp="useML_PDA" screensize="480x585"></filter>
    <!-- More filter options can be added -->
    <name>Useware</name>
    <!-- other UOs and eUOs have been deleted -->
        <useobject>
            <name>View operating statistics</name>
            <mapping>
                <machine typ="all"/>
                <user_group typ="supervisor"/>
                <location typ="all"/>
            </mapping>
            <select multiple_selection="false">
                <name>Time interval</name>
                <mapping>
                    <machine typ="all"/>
                    <user_group typ="supervisor"/>
                    <location typ="all"/> <function ID="T"/>
                </mapping>
                <option selected="true"><value>today</value></option>
                <option> <value>last day</value> </option>
                <option><value>last week</value></option>
                <option><value>last month</value></option>
            </select>
            <execute confirmation_required="yes">
                <name>Reset statistics</name>
                <mapping>
                    <machine typ="all"/>
                    <user_group typ="expert"/>
                    <location typ="local"/>
                </mapping>
            </execute>
        </useobject>
</usemodel>
```

Figure 11: A <use model> showing part of the task "View operating statistics"

As indicated in Figure 11 different filters can be associated with the model. These filters can then be used for generating prototypes for different devices. Figure 12 shows an example of two generated prototypes.

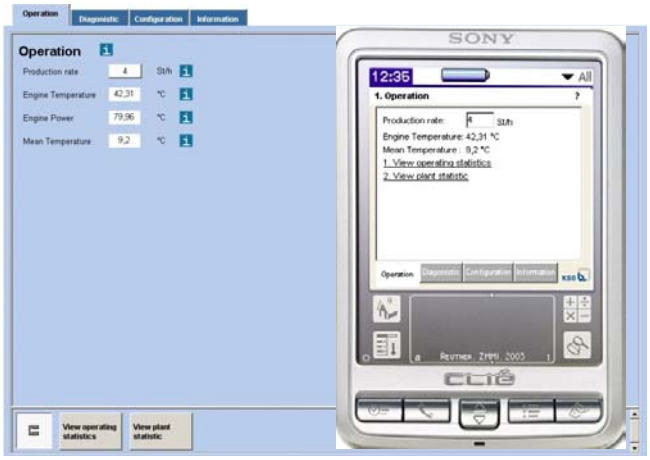


Figure 12: Useware prototypes for Web client and PDA

useML and the MB-UID

As proposed by [10] the Model-Based User Interface Description (MB-UID) approach enables the description of the user interface by using different declarative models at different levels of abstraction. At the highest level are the task model and domain model for the application. The task model represents the tasks that users need to perform, and the domain model represents the domain data to be manipulated, visualized or both, and the operations that the application supports. Task models typically represent tasks by hierarchically decomposing each task into sub-tasks (steps), until the leaf tasks represent operations supplied by the application.

The second level is the *abstract user interface specification* that represents the structure and content of the interface in terms *abstract interaction objects (AIO), information elements* and *presentation units*. AIOs are low-level interface tasks such as selecting one element from a set, or showing a presentation unit. Information elements represent data to be visualized. This can either be a constant value such as a label, or a set of objects and attributes drawn from the domain model. Presentation units are an abstraction of windows. They specify a collection of AIOs and information elements that should be presented to users as a unit.

The third level of the model, called the *concrete user interface specification*, specifies the style for rendering the presentation units, and the AIOs and information elements they contain. The concrete specification represents the interface in terms of toolkit primitives such as windows, buttons, menus, check-boxes, radio-buttons, and graphical primitives such as lines, images, text, etc. In addition, the concrete specification specifies the layout of all the elements of a window. While the abstract interface specification remains toolkit independent, its corresponding concrete model is toolkit specific.

Other models are the platform and user model.

useML implements some aspects of this approach by defining tasks and interactions. Though the <use model> basically defines tasks and interactions, it also defines an abstract presentation model in a platform independent way. In fact, the eUOs of the <use model> define abstract interaction objects. For example the eUO <select> can be mapped to a list, a combo box, a group of check boxes, a group of radio buttons or even a group of command buttons in a concrete presentation model. The decision of the presentation form to be taken will mainly depend on constraints of the hardware platform but also on the size of the selection values set and the semantic of their usage. While check boxes are normally used for multiple selections, radio buttons are for single selection. The two need more space on the display for realizing the selection. On the other hand, lists can be selected directly and offer an option for single or multiple selections. Thus one will tend to use buttons or check boxes if there is enough space on the display and list if space is limited.

RELATED WORK

A number of XML-based UIDL exist to date, some being for general purpose while others for UI in specific application domains. This section is a short discussion of some UDIL that are related to useML.

The XML-based User Interface Language (XUL)

XUL is a language for describing the contents and the layout of windows. To a certain degree cross-platform applications like web browsers and mail clients can be built from pre-defined libraries. A XUL user interface description is made up of three sets of files, each stored in a separate subdirectory of the chrome directory. These three sets are the content, the skin and the locale. A particular component might provide one or more default skins and locales, but a user can replace them with his own ones [13]. XUL elements are based on widgets like window, button, textbox hence it enables a detailed and concrete description of user interface for platforms supporting these widgets. XUL differs from useML in that it provides less or no abstraction and fits in the design and realization phases of the development process described above.

The User Interface Markup Language (UIML)

This is perhaps the most known XML-based UIDL. It provides platform independent elements for defining the user interface. The main elements are the <interface> that contains interface specific elements like <structure>, <content>, <style>, <behaviour> and the <peers> [11]. The <peers> element enables rendering the interface to a specific platform with platform specific toolkit. Though UMIL elements are platform independent, one needs platform knowledge in order to be able to define an interface for a specific platform. For example the developer must know if the structure he defines fits to the target platform. Therefore a UIML document is platform dependent. Taking useML und UIML together, it can be said that useML can provide UIML with a task-oriented and platform independent UI structure [4]. In this way UIML can fit in the design phase of the development

process shown in Figure 1. The relationship between useML and UIM is shown in Figure 13 below.

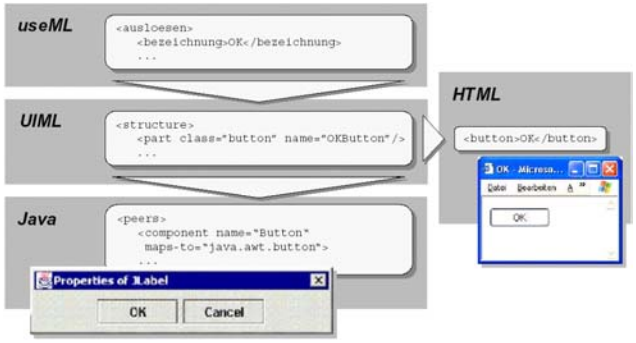


Figure 13: Combining useML and UIML

The Cooperative User Interfaces Markup Language (CUIML)

CUIML is a language for the generation of multimodal human-computer interfaces [7]. It is a XML dialect that is based on UIML. Its basic idea is to avoid code redundancy by putting parts that are common to all devices together, hence minimizing the effort for change. It also implements the MVC architecture that enables faster event handling. A CUIML document has <cuiml> as a root node with three mandatory children: <controller>, <genericHCI> and <rendering>. The <controller> node has one attribute, in which the URL of the WFDL document that configures the ControllerWFE is stored. The ControllerAFE embodies the central instance to synchronize the views and keep truck of the current state of the HCI at runtime. The <genericHCI> part describes the structure and behavior of the whole HCI. In the <rendering> section, parts are selected from this section and presented in different views.

Since CUIML is a UIML derivative, it is related to useML in the same way that UIML does.

The Interaction Markup Language (IML)

Instead of describing the user interface, IML first describes the interactions of the user with the interface without making any assumptions of its implementation. It separates the semantic information from the widgets and uses independent rules to describe the semantics. IML is intended to be 'appliance independent' i.e., it is not limited to certain devices or platforms. This appliance-independent XML description is transformed to specific user interfaces using a rendering engine, which generates the actual implementation [3].

IML resembles useML by abstracting user interaction instead of the user interface elements. This can guarantee platform independence, since user interactions are platform independent. One challenge of interaction abstraction is domain dependence. Generally interactions tend to be domain specific, meaning that general domain independent abstraction of interactions will be difficult to reach. useML was developed for machine Useware.

The eXtensible Interface Markup Language (XIML)

XIML defines necessary requirements for a universal user interface description language as well as its structure [5]. It proposes a solution for standardizing the representation and manipulation of *interaction data* – the data that defines and relates all the relevant elements of a user interface. In so doing, XIML is rather a framework than a description language. The basic interface components suggested by XIML have some resemblances with the models of the MB-UID [10]; the *task* component corresponds to the task model, the *domain* component corresponds to the domain model, the *user* component to the user model, the *presentation* component to the presentation model and the *dialog* component to the dialog model. The XIML Framework extends this by explicitly modeling relations between elements and their attributes by using the *relations* and *attributes* representational units. Though it is a good idea to standardize the representation and manipulation of interaction data, a concrete XIML-based implantation of the framework could not be found (at least it was not included in [5]). It is therefore not clear whether XIML as a language really exists.

A comparison of useML and the related work discussed here can be seen in Figure 14. Phases of the Useware development process that the UIDLs support have been used as comparison criteria to emphasize the relationship between the development process and the description language.

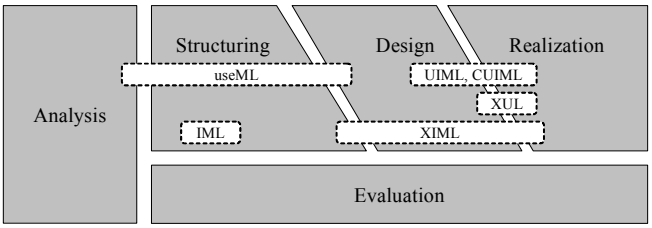


Figure 14: Mapping UIDL to the Useware development process

CONCLUSION AND FUTURE WORK

This paper has presented useML as a language for describing user interface for machines on XML basis. Its abstraction of user tasks and interactions by using <use objects> (UOs) and <elementary use objects> (eUOs) enables the description of a UI in a platform independent way. The eUOs types "execute", "select", "data input", "change" and "inform" abstract all interactions of a machine user. The instance of useML schema is the <use model> and can be treated with special style sheets to generate prototypes for different platforms, leaving the <use model> itself platform independent. The <use model> also represents an abstract presentation model of the MB-UID.

useML differs from other XML-based UIDL by being task oriented and "purely" platform independent. Due to its platform independency, useML can be applied to describe UIs even for devices that support new technologies like

ambient intelligence [1]. Considering the applicability of UIDL in the phases of the Useware development process, useML mainly supports the structuring stage, while most UIDL are developed for the design phase.

Tools for the transformation of the <use model> into design specific UIDLs like UIML are being developed at the institute. Moreover, approaches for integrating ergonomic design rules in the UIDL are also in progress. This will guarantee a consistent data flow in the Useware development process and simply the development process considerably.

REFERENCES

1. Boedcher, A. et. al. Organisatorisches Use model (Use model for distributes systems). *Technical Report pak,* University of Kaiserslautern, Kaiserslautern, 2004.
2. Constantine, L. et al.: From Abstraction to Realization: Canonical Abstract Prototypes for User Interface Design. Working Paper Version 2.0, http://www.foruse.com/articles/canonical.pdf, July 2003.
3. Göschka, K. M. and Smeikal, R.: Interaction Markup Language - An Open Interface for Device Independent Interaction with E-Commere Applications, in *Proceedings of the 34th Hawaii International Conference on Systems Sciences 2001. IEEE Computer Society,* Austria, 2001.
4. Mukasa, K., and Maidhof, K. User Interface Description Languages. *Technical Report pak.* Kaiserslautern University of Technology, Kaiserslautern, 2004.
5. Puerta, A., and Eisenstein, J.: XMIL: A Universal Language for User Interfaces, http://www.ximl.org/documents/XimlWhitePaper.pdf, April 2004.
6. Reuther, A. useML- Systematic Useware-engineering with XML. *Science-News pak, Vol. 8.* Kaiserslautern University of Technology, Kaiserslautern, 2003.
7. Sandor, C., and Reicher, T. CUIML: A Language for the Generation of Multimodal Human-Computer Interfaces, in *Proceedings of the European UIML*, 2001.
8. Stary, C.: Meeting Activity Theory through Task-Based and User-Oriented Development of User Interfaces, in *Proceedings of the Fourth International Conference on Computer-Aided Design of User Interfaces*, Valenciennes, May 2002, 193-204.
9. Style Guide for Machine Tools. A Handbook for Design of User Interfaces for Machine Tools, Fraunhofer IRB Verlag, Stuttgart, 2000.
10. Szekely, P. Retrospective and Challenges for Model-Based Interface Development, in *Computer-Aided Design of User Interfaces*, Belgium, 1996, xxi-xliv.
11. User Interface Markup Language (UIML) Specification. Language Version 3.0 (Draft), http://www.uiml.org/specs/uiml3/DraftSpec.htm, February 2002.
12. VDI/VDE3850 Norm: Nutzergerechte Gestaltung von Bediensystemen für Maschinen (user oriented design of machine control systems).
13. XUL Tutorial: http://xulplanet.com/tutorials/xultu/, February 2004.
14. Zühlke, D.; Mukasa, K.: Modelbasierte GUI-Entwicklung mit autoCAID. (Model-based GUI Development with autoCAID) in, *Qualität von Arbeit und Produkt in Unternehmen der Zukunft,* Stuttgart: Ergonomia Verlag, 2003, 91-94.
15. Zuehlke, D.; Wahl, M.: autoCAID: a model-based GUI tool for machine tools. IUI 2002, San Francisco, 2002, 242-242.

The TERESA XML Language for the Description of Interactive Systems at Multiple Abstraction Levels

Silvia Berti, Francesco Correani, Fabio Paternò, Carmen Santoro

{silvia.berti, francesco.correani, fabio.paterno, carmen.santoro}@isti.cnr.it
ISTI-CNR
Via G.Moruzzi 1
Pisa, Italy

ABSTRACT
The purpose of this paper is to report on the use of XML languages to support the TERESA tool. This is a tool for model-based design of multi-device interfaces. It considers three levels of abstractions (task model, abstract user interface and concrete user interface). For each of them a specific language has been defined and used. In addition, since the lowest abstract level (the concrete interface) is platform-dependent, there are different variants for each platform considered.

Keywords
Model-based design, XML user interface languages, tools.

INTRODUCTION
With the advent of the wireless Internet and the rapidly expanding market of smart devices, designing interactive applications supporting multiple platforms has become a difficult issue. The main problem is that many assumptions that have been held up to now about classical stationary desktop systems are being challenged when moving towards nomadic applications, which are applications that can be accessed through multiple devices from different locations. Consequently, one fundamental issue is how to support software designers and developers in building such applications: in particular, there is a need for novel methods and tools able to support development of interactive software systems that adapt to different targets while preserving usability.
Model-based approaches [4] could represent a feasible solution for addressing such issues: the basic idea is to identify useful abstractions highlighting the main aspects that should be considered when designing effective interactive applications. Our approach extends previous work in the model-based design area in order to support development of nomadic applications. In particular, we have designed and developed the TERESA (Transformation Environment for inteRactivE Systems representAtions) tool providing general solutions that can be tailored to specific cases. This tool supports transformations in a top-down manner, providing the possibility of obtaining interfaces for different types of devices from logical descriptions. It differs from other approaches such as UIML [1], which mainly consider low-level models. XIML [6] has similar goals but there is no publicly available tool supporting it.

THE METHOD
Our method for model-based design is composed of a number of steps that allow designers to start with an overall envisioned task model of a nomadic application and then derive concrete and effective user interfaces for multiple devices:

- *High-level task modelling of a multi-context application*. In this phase designers develop a single model that addresses the possible contexts of use and the various roles involved and also a domain model aiming to identify all the objects that have to be manipulated to perform tasks and the relationships among such objects. Such models are specified using the ConcurTaskTrees (CTT) notation [4], which also allows designers to indicate the platforms suitable to support each task.
- *Developing the system task model for the different platforms considered*. Here designers have to filter the task model according to the target platform and, if necessary, further refine the task model, depending on the specific device considered, thus, obtaining the system task model for the platform considered.
- *From system task model to abstract user interface*. Here the goal is to obtain an abstract description of the user interface composed of a set of presentations that are identified through an analysis of the task relationships. Each presentation is structured by means of interactors composed of various operators.
- *User interface generation*. In this phase we have the generation of the user interface. This phase is completely platform-dependent and has to consider the specific properties of the target device.

THE TOOL

TERESA is intended to provide a complete semi-automatic environment supporting a number of transformations useful for designers to build and analyse their design at different abstraction levels and consequently generate the user interface for various types of platforms.

A number of main requirements have driven the design and development of TERESA:

- *Mixed initiative*; we want a tool able to support different levels of automation ranging from completely automatic solutions to highly interactive solutions where designers can tailor or even radically change the solutions proposed by the tool.
- *Model-based*, the variety of platforms increasingly available can be better handled through some abstractions that allow designers to have a logical view of the activities to support.
- *XML-based*, each abstraction level considered can be described through an XML-based language.
- *Top-down*, this approach is an example of forward engineering. So, designers first have to create more logical descriptions, and then move on to more concrete representations until the final interface is obtained.
- *Different entry-points*, our approach aims to be comprehensive and to support various possibilities, including also when different set of tasks can be performed on different platforms. However, there can be cases where only a part of it needs to be supported and, for example, designers want to start with a logical interface description and not with a task model.
- *Web-oriented*, we decided that Web applications should be our first target. However, the approach can be easily extended to other environments (such as Java applications, Microsoft environments, …) by just modifying only the last transformation (from concrete interface to final interface).

The TERESA tool offers a number of transformations and provide designers with an integrated environment for generating XHTML interfaces for desktop, mobile phones and VoiceXML user interfaces. With the TERESA tool, at each abstraction level the designer is in the position of modifying the representations while the tool keeps maintaining forward and backward the relationships with the other levels. For example, it maintains links between abstract interaction objects and the corresponding tasks in the task model so that designers can immediately identify their relations. This results in a great advantage for designers in maintaining a unique overall picture of the system, with an increased consistence among the user interfaces generated for the different devices and consequent improved usability for end-users.

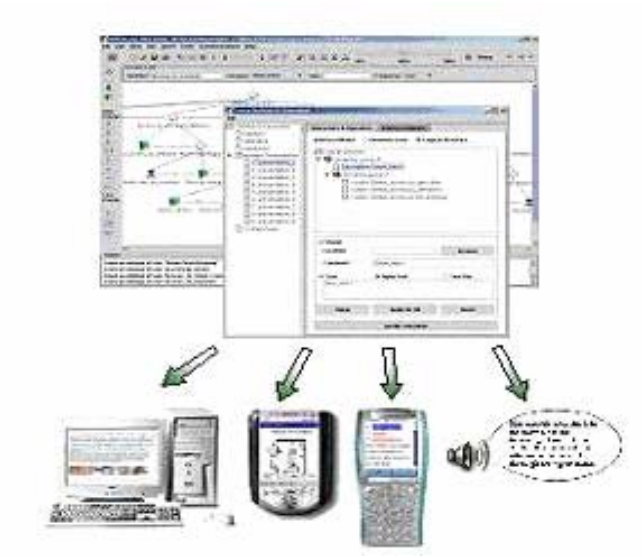


Figure 1: The TERESA tool.

Once the elements of the abstract user interface have been identified, every interactor has to be mapped into interaction techniques supported by the particular device configuration considered (characterised by the modalities supported, the screen size, …), and also the abstract operators have to be appropriately implemented by highlighting their logical meaning: a typical example is the set of techniques for conveying grouping relationships in visual interfaces by using presentation patterns like proximity, similarity and continuity. However, different techniques for grouping elements are used in case of vocal interfaces, such as using a specific sound to delimit a set of elements.

How XML-based Languages are Used in the Tool

TERESA is a transformation-based tool that supports the design of an interactive application at different abstraction levels and generates the concrete user interface for various types of platforms. By platform we mean a class of systems that share the same characteristics in terms of interaction resources. Such transformations exploit a number of XML languages. The main transformations supported in TERESA are:

- *Presentation task sets and transitions generation*. From the XML specification of a CTT task model concerning a specific platform, it is possible to obtain the Presentation Task Sets (PTSs), sets of tasks which are enabled over the same period of time according to the constraints indicated in the model and transitions specifying the conditions allowing moving across PTSs. Such sets, depending on the designer's application of a number of heuristics (general criteria used to merge together two or more PTSs) supported by the tool, can be grouped together so identifying the groups of tasks that should be supported by each user interface presentation.

- *From task model -related information to abstract user interface*. The goal of this phase is mapping the task-based specification of the system onto an interactor-based description of the related abstract user interface. Both the XML task model and Presentation Task Sets specifications are the input for the transformation generating the associated abstract user interface. Currently, each basic task that manipulates a user interface element is associated with an interactor in the abstract interface. The specification of the abstract user interface, in terms of both its static structure (the "presentation" part) and dynamic behaviour (the "dialogue" part), is saved for further analyses and transformations. It is worth pointing out that using TERESA it is also possible to access the inverse mapping, since for each interactor the tool is able to automatically identify and highlight the related task, so that designers can immediately spot such a relation. This is particularly useful especially when it comes to specifying the properties of each interactor, as the knowledge of the task it supports is an important indication of its meaning and goal, so it helps designers to position the interactor within the overall application and decide on the most appropriate settings.
- *From abstract user interface to concrete interface for the specific platform*. This transformation starts with the loading of a XML abstract user interface specification previously saved and yields the related concrete user interface for the specific media and interaction platform selected, which is saved in the associated XML language. Currently, there is a one-to-one mapping between abstract and concrete interactors. A number of parameters related to the customisation of the concrete user interface are made available to the designer.
- *Automatic UI Generation*. The tool automatically generates the final UI for the target platform. The starting point can be the single-platform task model, using a number of default configuration settings related to the user interface generation, or the abstract or the concrete user interface.

THE TASK MODEL

The task model is represented by the ConcurTaskTrees (CTT) notation [4], which supports a hierarchical description of task models with the possibility of specifying a number of temporal relations among them (such as enabling, disabling, concurrency, order independence, suspend-resume). In addition, for each task it is possible to specify what objects need to be manipulated for its accomplishment (it is possible to consider both user interface and domain objects), as well as a number of additional attributes (such as frequency) (see Figure 3).

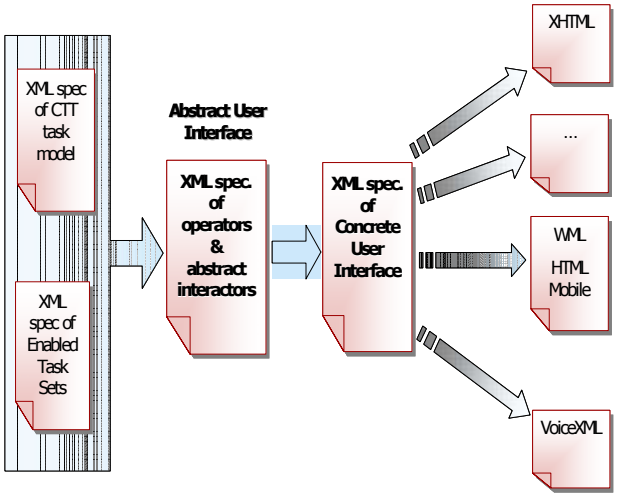


Figure 2: How XML Languages are used in TERESA.

The notation is tool supported. Initially the idea was to facilitate the development of task models and support saving them in XML format. It was the first notation for task models described in XML format (this feature has been available since 1998). It is currently used also in other environments (such as [2]) for user interface design and development.

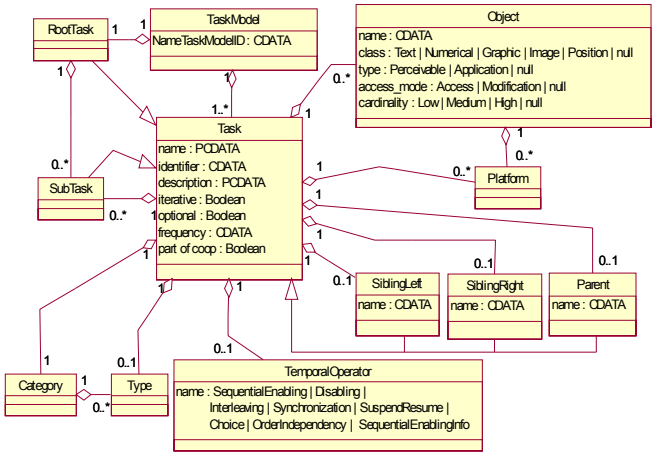


Figure 3: Class diagram representing the concepts of the ConcurTaskTrees notation.

In order to support this approach we needed to extend the notation in such a way to be able to capture the specific aspects of task models for nomadic applications. The platform attribute has been added in each task specification; its purpose is to indicate the types of platforms that are suitable to support it. It is worth noting that at this level –the task level- sets of devices sharing certain similarities are considered, rather than *specific* devices. So, in our framework we provide for typical sample device clusters as mobile phones and PDAs are,

together with the possibility for designers to define their own platforms. This has proved to be both feasible and flexible to tackle the problem of dealing with the disparate devices that our approach has to consider. Nevertheless, additional levels of refinement within the same cluster are considered in the last phase of the method, when knowing the specific characteristics of the devices considered becomes useful for producing effective final user interfaces.

The platform attribute has also been associated with the objects manipulated during task accomplishment. Indeed, CTT allows designers to specify for each task what objects should be manipulated during its performance.

THE ABSTRACT USER INTERFACE

An abstract user interface is composed of a number of presentations and connections among them. Each presentation defines a set of presentation and interaction techniques perceivable by the user at a given time. The connections define the dynamic behaviour of the user interface. More precisely, they indicate what interactions trigger a change of presentation and what the next presentation is. They can be associated with conditions in case a specific combination of interactions should trigger the change of presentation.

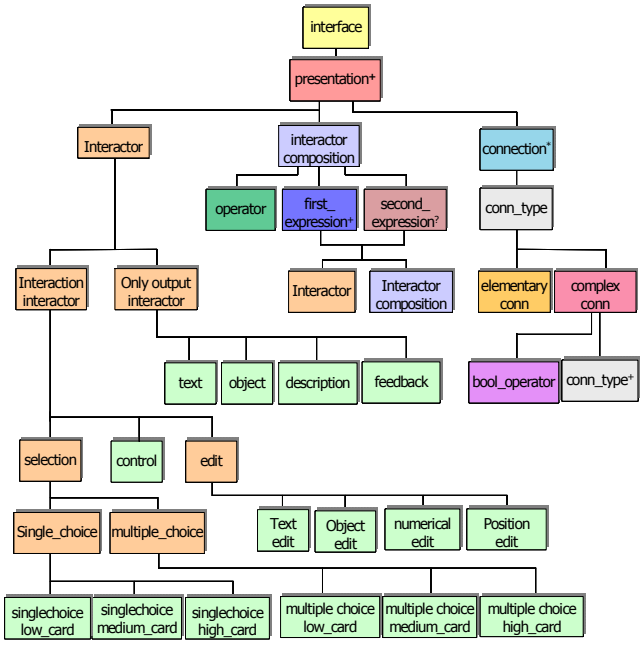


Figure 4: The Structure of the Abstract User Interface Language.

The structure of the presentation is defined in terms of interactors (abstract descriptions of interaction objects classified depending on their semantics) [5] and their composition operators (see Figure 4). It is possible to distinguish between interactors supporting user interaction (interaction elements) and those that present results of application processing (only_output elements). The interaction elements imply an interaction between the user and the application. There are different types of interaction elements depending on the type of task supported. We have selection elements (to select between a set of elements), edit (to edit an object), control (to trigger an event within the user interface, which can be useful to activate either a functionality or the transition to a new presentation). Differently, an only_output element defines an interactor which implies an action only from the application. There are different types of only_output elements (text, object, description, feedback) depending on the type of output the application provides to the user: a textual one, an object, a description, or a feedback about a particular state of the user interface.

The composition operators can involve one or two expressions, each of them can be composed of one, several interactors or, in turn, compositions of interactors. In particular, the composition operators have been defined taking into account the type of communication effects that designers aim to achieve when they create a presentation [3]. They are:

- Grouping (**G**): indicates a set of interface elements logically connected to each other;
- Relation (**R**): highlights a one-to-many relation among some elements, one element has some effects on a set of elements;
- Ordering (**O**): some kind of ordering among a set of elements can be highlighted;
- Hierarchy (**H**): different levels of importance can be defined among a set of elements.

CONCRETE GRAPHICAL INTERFACE

In this section we describe the XML language for two graphical platforms: the desktop and the mobile phone. In both cases the structure is designed to be similar to the structure of the abstract user interface language. In this way, passing through these levels in the UI generation process, we are always able to easily recognize the interactor hierarchy.

Implementation details are mostly provided by a deeper tree representation, with leaf nodes defining concrete information. Differences among concrete user interface specifications belonging to different devices, thus, can be mainly found in the lowest levels of the hierarchical structure.

A concrete desktop user interface is defined by a number of presentations and default settings to be used in the generation phase.

<!ELEMENT concrete_desktop_interface
(default_settings, presentation+) >

<!ELEMENT default_settings (background, font_settings, operators_settings, interactors_settings)>

Mobile devices also require specifying further data regarding expressive capabilities of the target phone.

<!ELEMENT concrete_mobile_interface (device_type, default_settings, presentation+)>

<!ELEMENT device_type (big | medium | small)>

<!ELEMENT big EMPTY>

<!ATTLIST big graphic_support (%option;) #REQUIRED>

<!ELEMENT medium EMPTY>

<!ATTLIST medium graphic_support (%option;) #REQUIRED>

<!ELEMENT small EMPTY>

<!ATTLIST small graphic_support CDATA #FIXED "no">

As usual, each presentation contains information about interactor organization and connections to other presentations, but specific properties, like title, header etc., are also described.

<!ELEMENT presentation (presentation_properties, connection*, (interactor | interactor_composition))>

<!ELEMENT presentation_properties (title, background, font_settings, top)>

Connection and interactor composition are defined as in the AUI language for each type of target platform.

<!ELEMENT connection (conn_type)>

<!ATTLIST connection presentation_name IDREF #REQUIRED>

<!ELEMENT conn_type (elementary_conn | complex_conn)>

[…]

<!ELEMENT interactor (interaction | only_output)>

<!ELEMENT interactor_composition (operator, first_expression+, second_expression?)>

<!ELEMENT operator (grouping | ordering | hierarchy | relation)>

In the Interactor specification the deeper we proceed, the more concrete details we get. For instance, let us consider how we can describe single selection interactors with our desktop CUI notation.

<!ELEMENT interaction (selection | editing | control)>

<!ELEMENT selection (single | multiple)>

<!ELEMENT single (radio_button | list_box | drop_down_list)>

<!ATTLIST single cardinality (%cardinality_value;) #REQUIRED>

<!ELEMENT radio_button (choice_element+)>

<!ATTLIST radio_button label CDATA #REQUIRED>

<!ELEMENT choice_element EMPTY>

<!ATTLIST choice_element

label CDATA #REQUIRED

value CDATA #REQUIRED>

[…]

Differences among other CUIs consist of different concrete elements associated to a given type of interactor; e.g. let us consider the previous example for mobile devices, list boxes are not usable in this context so they are not allowed.

<!ELEMENT single (radio_button | drop_down_list)>

[…]

Each CUI formalizes the expressive power of a given device type in terms of concrete interactors and operators available in that platform. While in desktop environments grouping operators can be implemented by combining several techniques, in mobile phones, because of the limited screen dimensions, we can choose only one implementation technique from a limited set.

It can happen that some abstract interactor is related to the same concrete elements even in different CUI. In this case, differentiation is granted by allowing different attributes values for the same concrete object.

For instance we can refer to the text edit objects: they can be implemented with a text field in both desktop and mobile cases.

<!ELEMENT text_edit (textfield)>

<!ELEMENT textfield EMPTY>

<!ATTLIST textfield

label CDATA #REQUIRED

length (%length_value;) #REQUIRED

password (%option;) #REQUIRED>

However, the reduced screen capabilities of mobile devices are considered, thus allowing lower length values.

Desktop CUI

<!ENTITY % length_value "8 | 9 | 10 | 11 | 12 |13 | 14 | 15 | 16 | 17 | 18 |19 | 20">

Mobile CUI

<!ENTITY % length_value "4 | 5 | 6 | 7 | 8 | 9 | 10">

CONCRETE VOCAL INTERFACE

Like graphical interfaces, in vocal interfaces a concrete user interface is composed of some default settings and a set of presentations corresponding to the presentations of the abstract user interface language. The difference is that in this language, interactors and their compositions are obtained through techniques specific for the vocal interface, so they have different attributes.

The default settings are applied to the entire vocal application and are important for supporting the user interactions.

<!ELEMENT default_settings (name_application, welcome_msg, def_commands?, synthesis_properties, recognition_properties, bargein, operator_settings+)>

<!ELEMENT name_application EMPTY>

<!ATTLIST name_application value CDATA #REQUIRED>

The welcome message allows users to understand the current context and that they are talking to a computer that accepts a well defined language. In this case is possible to use some default message (short, medium or long) or to define a new message (new). Another useful parameter of this element is onlyOnce that allows skipping a welcome message when the user visits the main presentation for the second time.

<!ELEMENT welcome_msg EMPTY>

<!ATTLIST welcome_msg

type (short | normal | long | new) #REQUIRED

msg CDATA #REQUIRED

onlyOnce (%boolean;) #REQUIRED>

Other general settings regard: the property of synthesis or recognition engine, the barge-in option that allows the user to interrupt a prompt in order to speed up the dialog sequence and some default commands that allow users to disable the device input and/or to exit from voice application.

<!ELEMENT synthesis_properties EMPTY>

<!ATTLIST synthesis_properties

pitch (%pitch_value;) #REQUIRED

rate (%rate_value;) #REQUIRED

volume(%volume_value;) #REQUIRED>

<!ELEMENT recognition_properties EMPTY>

<!ATTLIST recognition_properties

confidence CDATA #REQUIRED

sensitivity CDATA #REQUIRED
completetimeout CDATA #REQUIRED
incompletetimeout CDATA #REQUIRED>

<!ELEMENT def_commands (exit?, disable?)>

<!ELEMENT disable EMPTY>

<!ATTLIST disable

cmd_dis CDATA #REQUIRED

cmd_activ CDATA #REQUIRED>

<!ELEMENT exit EMPTY>

<!ATTLIST exit msg_to_exit CDATA #REQUIRED>

<!ELEMENT bargein EMPTY>

<!ATTLIST bargein active (%boolean;) #REQUIRED>

The structure of each presentation is defined in terms of some general properties similar to those considered in the default settings but in this case they concern one specific presentation, the dynamic behaviour of the user interface, the vocal properties of interactors, and the vocal techniques exploited to represent the composition operators.

<!ELEMENT presentation (presentation_properties, connections*, (interactor | interactor_composition))>

<!ATTLIST presentation name ID #REQUIRED>

For example, the vocal properties of selection interactor concerns three types of menu:

<!ELEMENT selection (single | multiple)>

<!ELEMENT single (dtmf_menu | enumerate_menu | message_menu)>

<!ELEMENT multiple (dtmf_menu | enumerate_menu | message_menu)>

The meaning of the possible values is:

- <dtmf_menu>: in this case in the vocal interface the user can perform the selection only through the keypad and so the message of synthesizer will be “If you want a coffee, press 1; if…”
- <enumerate_menu>: in this case in the vocal interface the synthesizer produce a list of option as “Do you want: coffee, tea, milk…”.
- <message_menu>: in this case in the vocal interface the synthesizer generate an elaborated message like “if you prefer coffee, say coffee; if…”

In any case, the selection interactors can define a feedback message to confirm if a command is correctly understood or how to manage some events such as no input or no match or help or define a reply message in order to listen again the choices.

In the concrete user interface composition operators can be represented through different vocal techniques according to their logical meaning and communication goals. Grouping can be represented through four techniques:

<!ELEMENT grouping (Insert_sound | Insert_pause | Change_volume | Keywords)>

<!ELEMENT Insert_sound EMPTY>

<!ATTLIST Insert_sound

src_audio_file CDATA #REQUIRED>

The technique inserts a sound at the beginning and at the end of the grouped elements and, in this case, the audio file is specified.

<!ELEMENT Insert_pause EMPTY>

<!ATTLIST Insert_pause

lenght_pause CDATA #REQUIRED>

This technique inserts a pause at the end of the grouped elements and in this case is specified the duration of pause.

<!ELEMENT Change_volume EMPTY>

A specific volume can be used during the speech synthesis of the grouped elements.

<!ELEMENT Keywords EMPTY>

The keywords technique inserts some words to highlight the grouping operator (for examples: "In this Application you can choose one of this option: If you would like some general information, say information… if you would like to book a ticket, say ticket. Alternatively if you would…").

Ordering can be represented by two techniques: arranging objects in alphabetical order, and keywords techniques that insert some words to highlight the operator order (for example: "In this presentation at the beginning you should say a name, after the sound say password and lastly say go in order to proceed").

<!ELEMENT ordering (Arrange_objects_in_alphabetical_order | Keywords)>

<!ELEMENT Arrange_objects_in_alphabetical_order EMPTY>

<!ELEMENT Keywords EMPTY>

The Relation operator supports a vocal input that enables a change in context by moving to another presentation. This type of operation can be useful to navigate within the presentation.

<!ELEMENT relation (Change_context)>

<!ELEMENT Change_context EMPTY>

The Hierarchy operator is represented through two techniques: increasing or decreasing the volume of the synthesized voice.

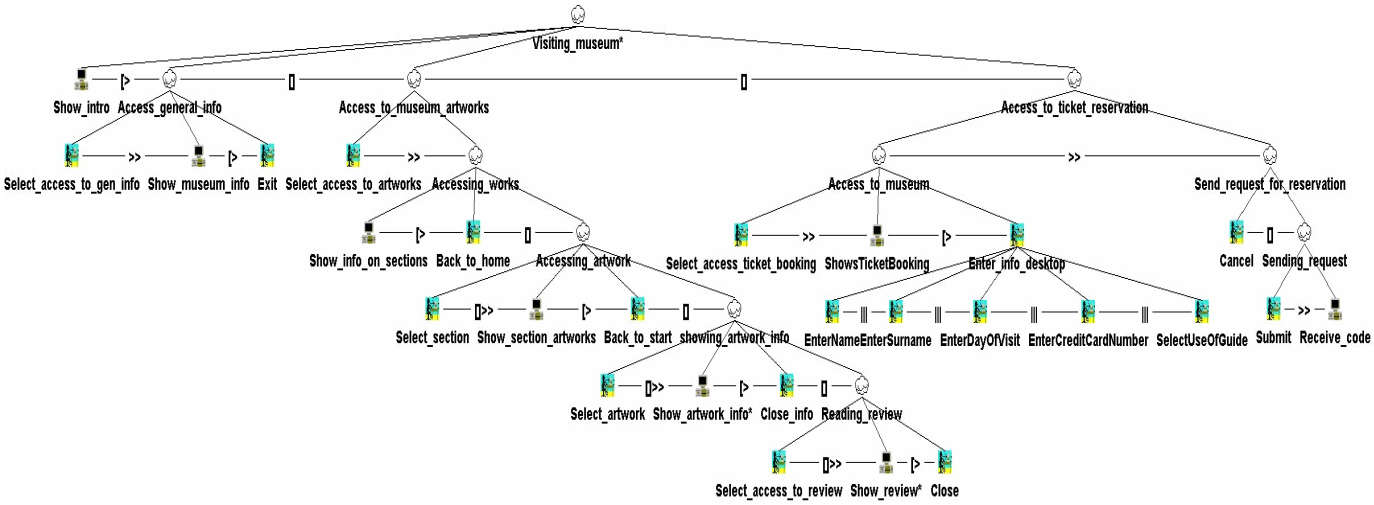


Figure 5: The Task Model for the Desktop version of the example.

AN EXAMPLE

In this section we show an example of this approach in order to better understand how the various abstraction levels are exploited. We consider a museum application.

Figure 5 shows the task model for the desktop version. After the automatic transformation of the task model first into an abstract user interface and then into a concrete user interface, some attributes have been edited in order to obtain the concrete version for a desktop system and we obtain the result shown in Figure 6.

It is possible to see that the user interface is structured into nine presentations automatically identified. The structure of the first presentation is currently presented by the tool. There are two grouping compositions of interactors: one aims to group three navigator interactors allowing the access to the various parts of the application, the second

one groups the first grouping with a description interactor. The element currently selected in the control panel is the second grouping interactor. The associated attributes and the corresponding values are displayed in the bottom part.

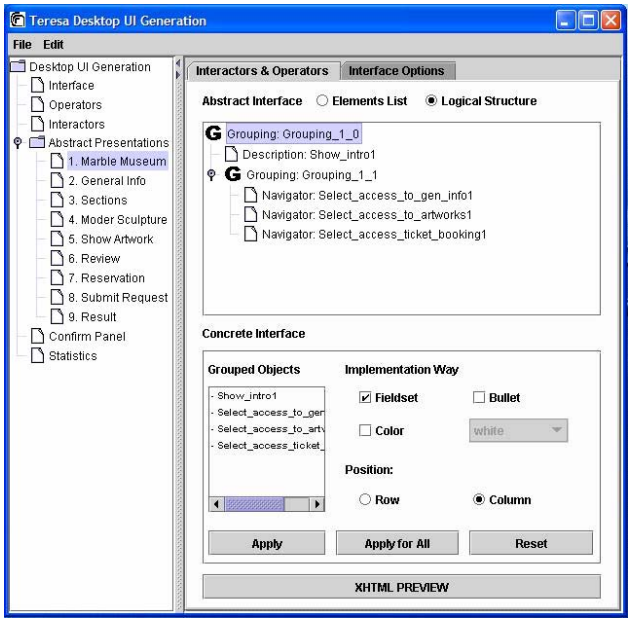


Figure 6: The Structure of the User Interface of the example.

Figure 7 shows the corresponding user interface. In the bottom part we can see the navigator interactors which are lined up in a horizontal manner with consistent appearance, thus implementing the grouping operator. In the top part there is the implementation of the description element aiming to introduce the Marble Museum and its artworks. The resulting interface implements the indications contained in the task model. Indeed, in the model the first is a system task aiming to introduce the museum showing the home page. Such task can be disabled by three interactive tasks whose purpose is to enable different types of high-level tasks (access to general information, artworks, and ticket reservation). These four basic tasks are those associated with the first presentation in the abstract and concrete user interface.

CONCLUSIONS

The TERESA environment supports design and development of multi-platform user interfaces through a number of transformations that can be performed either automatically or through interactions with the designer.

To this end, a number of XML languages that capture the relevant information at different abstraction levels are used. Such languages are introduced in this paper along with a discussion of how they are used in the environment.

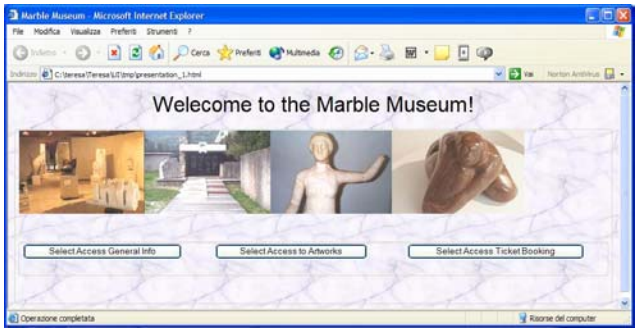


Figure 7: The interface of the first page of the example.

The tool can be freely downloaded at http://giove.cnuce.cnr.it/teresa.html.

Future work will be dedicated to supporting generation in further multimodal user interface languages. We also plan to support importing of CAMELEON XML representations in order to ease exchange of information with other tools and facilitate integration of forward and reverse engineering environments

ACKNOWLEDGMENTS

We gratefully acknowledge support from the EU IST CAMELEON project (http://giove.cnuce.cnr.it/cameleon.html) and the EU SIMILAR NoE (www.similar.cc).

REFERENCES

1. Abrams, M., Phanouriou, C., Batongbacal, A., Williams, S., Shuster, J. *UIML: An Appliance-Independent XML User Interface Language*, Proceedings of the 8th WWW conference, 1999.
2. K.Luyten, K.Conix, An XML-based runtime user interface description language for mobile computing devices. Proceedings DSV-IS 2001, pp.20-29, Springer Verlag.
3. Mullet, K., Sano, D., Designing Visual Interfaces. Prentice Hall, 1995.
4. Paternò, F., Model-Based Design and Evaluation of Interactive Application. Springer Verlag, ISBN 1-85233-155-0, 1999.
5. Paternò, F., Leonardi, A. A Semantics-based Approach to the Design and Implementation of Interaction Objects, Computer Graphics Forum, Blackwell Publisher, Vol.13, N.3, pp.195-204, 1994.
6. Puerta, A., Eisenstein, XIML: A Common Representation for Interaction Data, Proceedings ACM IUI'01, pp.214-215.

VRIXML : A User Interface Description Language for Virtual Environments

Erwin Cuppens **Chris Raymaekers** **Karin Coninx**

Limburgs Universitair Centrum
Expertise centre for Digital Media
Universitaire Campus, B-3590 Diepenbeek, Belgium
{erwin.cuppens, chris.raymaekers, karin.coninx}@luc.ac.be

ABSTRACT

When modeling virtual environments, the designer has to consider the interaction of the user within the generated world, next to the appearance of the environment. Part of this interaction is supported by a user interface, which can be used to manipulate the objects that are contained in the environment.

This paper investigates the possibility of using a XML-based user interface description language in order to design a user interface and specify its behavior within the virtual environment. Some existing languages are examined and compared, and afterwards the syntax and features of our own description language, called VRIXML, are presented.

Author Keywords

Virtual Environment, User Interface Description Language

INTRODUCTION AND MOTIVATION

Virtual Environments (VEs) are computer generated, three-dimensional environments that create the effect of an interactive world in which the user is immersed. Nowadays, VEs can be used in a lot of different application areas such as Computer Aided Design (CAD) and 3D modeling [17], (medical or flight) simulation environments [18], and information visualization [3].

Virtual environments are very promising for future computer science applications but, like every maturing technique, there still are some problems to be solved before VEs become more widespread. The most important challenge when using computer generated environments, is to keep the feeling of presence at all times. One way to realize this for the user is the use of advanced display techniques such as stereoscopic viewing, head mounted displays and cave structures. A second aspect that contributes to this feeling are the interaction paradigms. The user is offered a range of techniques that allow intuitive interaction with the generated environment. Some examples of these interaction techniques are speech input, haptic feedback or 3D (head)tracking. Advanced use of these techniques leads to two-handed and/or multimodal interaction as we use it in real life [4].

The expansion of the existing interaction possibilities provides particularly methods for direct manipulation of the objects in the environment. When considering for instance an object that is positioned at a large distance of the user, it is easy to see that direct manipulation is not always obvious. To tackle this problem, indirect manipulation is required. A straightforward approach to solve the need for indirect manipulation techniques is to integrate a (two-dimensional) GUI in the three-dimensional environment for object manipulation. A hybrid 2D/3D user interface is a possible approach in order to provide this interface integration. [5]

In our lab we are developing a code framework that generalizes interaction in VEs. The framework contains an elaborated widget set for creating user interfaces in a computer generated environment. The design of the widget set is based on previous research and it will be presented more in detail in the section concerning the user interface and its environment.

Nowadays, XML-based user interface description languages (UIDLs) are frequently used to design an application interface, particularly for desktop, web-based and multi-device user interfaces. The last few years, a lot of research has been done towards these UIDLs and several languages are presented for all kinds of purposes.

In the scope of our current research project, which is called VR-DeMo (Virtual Reality: Conceptual Descriptions and Models for the Realization of Virtual Environments), we are examining how we can use a XML-based UIDL in order to design our user interface and connect it to the functionalities of the VE. For this reason we created VRIXML (Virtual Reality Interaction XML).

The ultimate goal of the VR-DeMo project is to facilitate and shorten the development process of VEs by means of conceptual specifications and descriptions. We believe that the use of a UIDL is a good first step in our research because of its descriptive nature.

In the remainder of this paper, some related work will be

presented after which an overview of the created framework and the hybrid widget set will follow. Next, the widget set will be connected to the developed description language of which the main properties will be illustrated with an example. Finally we will conclude the paper by presenting some ideas for the future.

RELATED WORK

User interface development in VEs has known several different approaches. Some limit the interface to menu interaction such as the spherical menu [11] or pie menu [7]. The first is also called a "Daisy menu" and is part of the JDCAD system which is used to manipulate 3D objects. The primitives are selected by rotating the sphere shaped menu until the desired object is in a cone that always faces the user. The pie menu is part of the Holosketch VR sketching system. The menu items are slices of the pie and are selected by means of a wand. If the selected menu item contains a sub-menu, the current pie fades out and the sub-menu pie fades in.
In other approaches (e.g. the FLIGHT project), complete toolkits were developed to create user interfaces for VEs [2].
Because of the more extended possibilities we prefer to develop a complete widget set over a solution with menus alone. Next to the widget set, there is also need for a way to describe the user interfaces that are developed.
eNode UI [8], libGlade, and WML [10] are languages that can be used to describe static interfaces. Each of these languages however, is designed to support a different type of interface and/or platform. eNode UI aims especially at the creation of user interfaces for (java-based) web applications. libGlade is a library allowing to load GLADE (a free user interface builder for GTK+ and GNOME) interfaces at runtime in such a way, that changing the look of a user interface is possible without recompiling the application. WML(wireless markup language) is an alternative for (X)HTML which is used to design webpages for mobile devices.
VoiceXML [9] is a language which has been designed in order to create audio dialogs that feature synthesized speech, digitized audio, ...
The SEESCOA XML [13] is a description language created, in our research lab, in the scope of the SEESCOA project. Its main goal is to describe user interfaces for embedded systems and mobile devices. An abstract description of the interface widgets was used, which offers possibilities for automatic user interface generation in such a way that the interface is adjusted to the constraints of the system (e.g. the size of the screen or the available widget set).
Another widely used description language is UIML (User Interface Markup Language). This is a more abstract XML-based meta-language that permits a declarative description of a user interface in a highly device independent manner [1]. Next to the description of the interface, UIML also offers the possibilities to map the defined interface to the application logic of the target device. Because of it's generality, it is possible to use UIML to as a replacement for other existing description languages such as VoiceXML, or WML. But, as always, abstraction has it's price and in the case of UIML, the drawback is complexity.
As far as we know, there are no UIDLs that focus on interaction in VEs. Compared with two-dimensional interfaces there are however some pitfalls, which make most of the existing languages less usable. An example of such a difficulty is the use of a more advanced widget set which is not supported in most UIDLs.
Next to the interface description languages, there are some languages that can be used to describe objects in virtual environments. Currently the dominant description language to describe virtual worlds, is VRML [12] (or X3D, which is VRML translated into an XML-based syntax). This language is widely used to define three-dimensional objects and their environments. Several viewers are created that use VRML files to generate three-dimensional worlds. Within these viewers, interacting with the environment is possible by means of a scripting language (e.g. javascript) but the interaction is limited to a mouse or a keyboard. Currently, no applications exist that support interaction in VRML while using VR input devices.

Because description languages have proven to be a very good method for the development and abstraction of 2D interfaces, we propose a similar approach for the description of user interfaces in virtual environments.

THE USER INTERFACE AND ITS ENVIRONMENT

Our lab has conducted several research projects towards interaction in VEs [4, 16, 17]. Based on these research results, we have developed a code framework that supports all investigated interaction techniques. In the remainder of this section an overview of the framework will be given. Special attention is payed to the user interface widget set that is supported.

The VE framework

A full description of the technical details and the goals of the framework is beyond the scope of this paper. For this reason we will cut down our overview to the properties of the framework necessary for the further understanding of this paper. Interested readers can find an extended description of the framework in [6]
The framework, in its current state, can be considered as a black box (indicated by the dashed box in figure 1) that can be used to create VEs that require all sorts of (multimodal) interaction possibilities. Currently it supports the use of several 3D input devices (e.g. a spacemouse, a microscribe or 3D trackers), speech input and haptic feedback by means of a PHANToM device (figure 2)[15]. However, due to the nature of this paper, interaction techniques, other than the user interface, will not be considered more in detail.
In order to reuse the interaction techniques, implemented within the framework, we tried to abstract these techniques as much as possible. To realize this, an in-

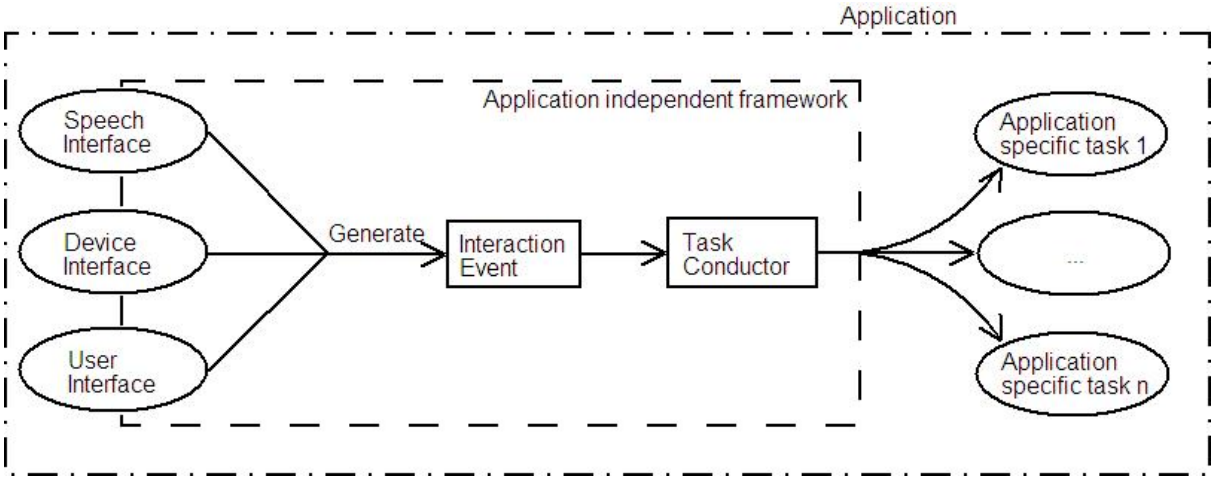


Figure 1: The VE Framework

terface is developed for each of these interaction techniques. These interfaces should be considered in the sense of an API of which the designer of the environment can use the supported functionalities.
The data passed into the framework is similar for all created interfaces and is represented by an *interaction event*. Such an event contains, besides the event identifier, all parameters necessary for the correct execution of the task. All events that are generated by the interaction techniques are send to a central dispatching unit, called a *task conductor*, located in the core of the framework. From there, the interaction events are redirected to the appropriate (application specific) event handling code as shown in figure 1.
In the remainder of this section an overview will be given of the provided user interface widget set. Like other interaction techniques, the user interface will also generate and send interaction events to the task conductor in order to perform the necessary tasks.

The user interface widget set

A hybrid approach was chosen when developing our widget set. The resulting user interface contains properties based on 2D as well as 3D environments and applications. As it was the main goal to be integrated in a VE, the interface can be controlled by several of the aforementioned 3D interaction devices. On the other hand, the widget set has a similar look and functionality as familiar 2D GUIs and the interaction was designed to be consistent with the skills of the users to work with 2D interfaces.
A special feature of the designed interface elements is their possibility to support haptic feedback by means of the PHANToM haptic interface [15]. This device provides the user with a sense of touch in the VE. The addition of this modality into a computer generated world can greatly increase the feeling of presence and improve the intuitiveness [19]. By combining haptic feedback with our widget set, user interface elements can be accessed in a more convenient way. Indeed, by resting the PHANToM pointer against the elements surface, the users movement is limited to 2 dimensions, similar to a 2D GUI.

Figure 2: The PHANToM haptic interface

In this paper we elaborate on a subset of the classic 2D widget set. This subset supports most interaction possibilities and is sufficient to prove the usefulness of our approach.
A user interface, based on our widget set, can be constructed using 3 types of container elements, namely a menu, a toolbar or a dialog. The menu, shown in figure 4a, and the toolbar provide nearly the same possibilities. Both can be used to trigger certain events or to open a sub-menu/sub-dialog. The menu consists of string items, where the toolbar usually contains buttons (with graphical icons). The dialog is a more extended container which can contain all other elements of the widget set, as shown in figure 3. By using the result values of these subwidgets, a dialog can provide more complex input into the system.
We have to draw the attention to the fact that it is generally not trivial to acquire character data in VEs, due to the lack of good text input techniques. If the user is using a head mounted display to view the environment, the keyboard cannot be used and even when the user

is not completely immersed in the environment, interaction often will require two-handed input techniques. Other input techniques such as speech input or a virtual keyboard are still not mature and accurate enough. For these reasons we choose not to implement widgets to support text input.

USER INTERFACE DESCRIPTION

After presenting part of the developed framework and the user interface in the previous section, this section will focus on the syntax and features of XML-based description languages. Special attention will be payed to VRIXML, the language we developed to describe our widget set.

During the last few years, several user interface description languages have been presented such as eNode UI, libGlade, RIML, SEESCOA XML, UIML, XIML ...

In this section we will briefly describe some of these languages and we will discuss why we chose to develop our own description language. Afterwards the syntax and the features of our language will be illustrated with some examples. Finally we will discuss briefly some of the differences in specifying 2D and 3D user interfaces.

A UIDL overview

Several of the existing languages are designed to describe a static user interface for specific application types (e.g. eNodeUI for web-applications) or platforms (e.g. libGlade for GLADE interfaces). Others, such as VoiceXML, are designed to serve very specific goals and are hard to use in other kinds of applications.

The language we are aiming for has to be more than only a description of the user interface. Our goal is to create a syntax that provides possibilities for the designer to specify certain application functionalities through the interface description.

Although several of the enumerated description languages such as SEESCOA XML and UIML, already have tackled this problem, we still choose to design our own language for several reasons.

SEESCOA XML is a description language designed in the scope of the SEESCOA project. The language is mostly used to describe user interfaces for embedded systems and mobile devices. The idea behind its syntax is comparable to what we are aiming for, that is: a strict syntax with a limited complexity. However, it is quite hard to match the syntax of the SEESCOA XML with the elements of our widget set and therefore, the effort of creating a new UIDL is similar, maybe even smaller, than extending the current SEESCOA XML.

UIML on the other hand is a very general language which covers a very wide range of application possibilities. It is very likely that our syntax, which will be illustrated in the next section, can be translated into an UIML syntax quite easy. However, because of its generality, UIML is less strict as we would like it to be. An XML file with an explicitly defined content can easily be parsed and validated against a document type definition (DTD) or SCHEMA.

The validation of a UIML file is slightly more complex because next to its syntax, the content has to be defined in a separate document, called a vocabulary, and at this time, there are no tools that support the validation of an UIML file against its vocabulary.

Basic description language syntax

Listing 1 shows an extract of the XML file in which the object properties dialog (figure 3) is described. We will use this short code fragment to point out some of the properties of the designed language.

```
<UIDialog>
  <Texture>
    <Name>tex_Properties.png</Name>
    <Color R="1.0" G="1.0" B="1.0"/>
  </Texture>
  <Title>Object Properties</Title>
  <DialogItem>
    <UIGroup>
      <GroupItem>
        <UIStatic>
          <Text>Diffuse Color</Text>
        </UIStatic>
        <Position>
          <X>1.0</X><Y>0.0</Y>
        </Position>
      </GroupItem>
      <GroupItem>
        <UIStatic>
          <Text>R</Text>
        </UIStatic>
        <Position>
          <X>0.0</X><Y>1.85</Y>
        </Position>
      </GroupItem>
      <GroupItem>
        <UISlider paramID="10">
          <Value min="0" max="255"/>
          <Tickstyle
              orientation="horizontal"
              position="both"
              frequency="16"/>
        </UISlider>
        <Position>
          <X>1.0</X><Y>1.5</Y>
        </Position>
      </GroupItem>
      ...
      <Event>14</Event>
    </UIGroup>
    <Position>
      <X>0.0</X><Y>0.0</Y>
    </Position>
  </DialogItem>
  ...
  <DialogItem>
    <UIGroup>
      <GroupItem>
        <UIButton>
          <Text>CANCEL</Text>
        </UIButton>
        <Position>
          <X>0.0</X><Y>0.0</Y>
        </Position>
      </GroupItem>
      <Event>0</Event>
    </UIGroup>
    <Position>
      <X>10.0</X><Y>15.0</Y>
```

```
    </Position>
  </DialogItem>
  <Position>
    <X>0.0</X><Y>0.0</Y><Z>-20.0</Z>
  </Position>
  <Metrics
      Horizontal="middle"
      Vertical="middle"/>
</UIDialog>
```

Listing 1: Extract of the ObjectPropertiesDialog.xml

As appears from the code, each widget of the user interface has its own specific tag, containing attributes and/or subtags in order to specify all properties of the element.
A dialog for example has some subtags to define its title, a texture for the titlebar, and its position in the virtual world. Next to those specific properties, the dialog also consists of one or more dialog items. Each dialog item contains a group of items and has a relative position within the dialog.
All widgets, contained in the same dialog, that have complementary functionalities (e.g. the checkbox to indicate whether a texture should be used and the combobox to specify the texture), should be described as items of the same group in order to facilitate the automatic event generation process (which will be explained more detailed in the following section).
If an element value should be passed as a parameter of the event, generated by its group, the user should add the optional parameter ID to the XML-tag which describes the element.

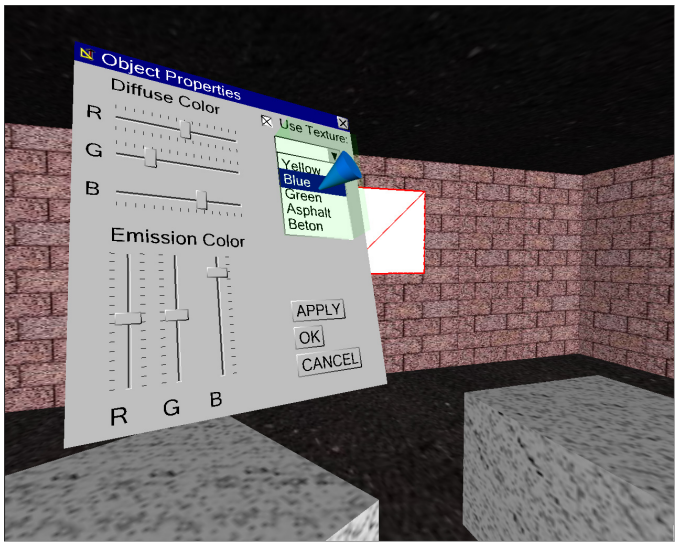


Figure 3: An Object Properties Dialog

Features

We will now present two of the main features of VRIXML and illustrate these with an example. At first we show how the created description language supports the automatic event generation process. Secondly, the (partially) automatic inter-dialog navigation of the framework will be presented and related to the user interface description.

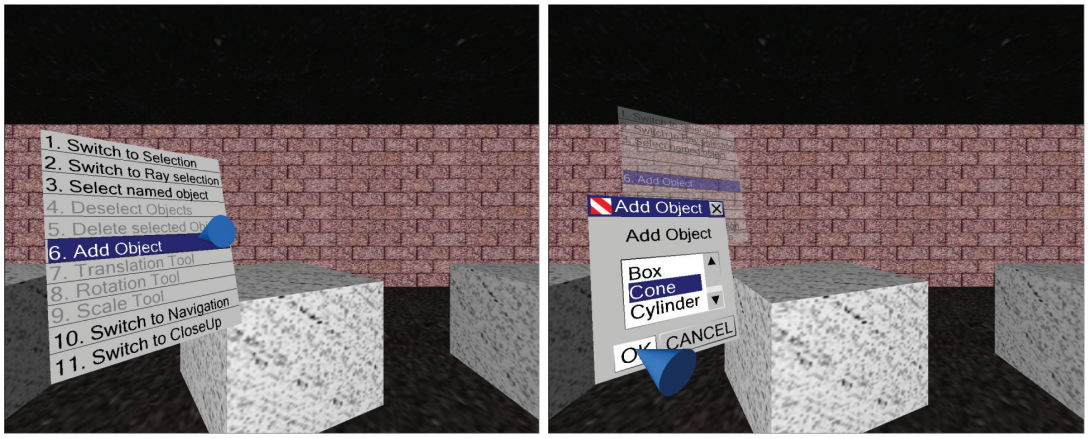
Description of event generation

As stated in the presentation of the framework, each interaction of the user with the VE generates an interaction event, in order to handle all similar events in a single task, independent of the event provider. Because a user interface is an alternative method to manipulate the virtual world, it is also necessary for the interface to generate these events.
Our description language facilitates the generation of these events by allowing the user to specify the events, with their parameters, within the user interface description. As a result of this, the user does not have to worry about how the events are generated. It is sufficient to group all complementary widgets, specify their parameter IDs, and specify the event which they belong to.
Consider the example of figure 3: when the OK button of the object properties dialog is triggered, all groups in the dialog are iterated and if a group specifies an event ID (e.g. `<Event>14</Event>` which is specified as the *SetObjectDiffuseColor* event), a new interaction event is generated. Next, a second iteration starts, in which the values, of all group items that specify a parameter ID (e.g. `<UISlider paramID="10">` which is specified as the *RColor* parameter), will be added to the newly generated interaction event. Once this second iteration has ended and all parameters are added, the event will be sent to the task conductor for further treatment, while the first iteration continues until no further events have to be generated.
The identifiers used for the events and their parameters are defined in an XML file that is accessible through the entire application.

Description of inter-dialog navigation

Next to automatic event generation, our framework also supports (partially) automatic inter-dialog navigation. In listing 2 is shown part of the XML description of the explore menu (Figure 4(a)). Next to event IDs, a menu item can also specify the ID of an other interface element (e.g. `<ChildUI>52000</ChildUI>` which specifies the Add Object dialog). When such a menu is triggered, instead of generating an event, the framework will automatically open the interface element, specified by the ID. The new interface element appears at the position of the menu while the latter is animated to the background (as shown in figure 4(b)). Once the menu moves to the background, the last chosen item will stay selected and for now, the menu cannot be used for interaction purposes. Its only importance is to allow the user to keep track of his interface navigation at all times. This approach is ideal to prevent the user of losing his/her sense of orientation [20]. When the dialog is closed, the menu animates back to its old position.

```
<UIMenu>
  ...
  <MenuItem ID="5">
    <Text>5. Delete selected Object</Text>
    <Event>17</Event>
```



(a) The Explore Menu (b) The Add Object Dialog

Figure 4: Inter-dialog navigation

```
  </MenuItem>
  <MenuItem ID="6">
    <Text>6. Add Object</Text>
    <ChildUI>52000</ChildUI>
  </MenuItem>
  ...
  <Position>
    <X>0</X><Y>0</Y><Z>-20</Z>
  </Position>
  <Metrics
      Horizontal="middle"
      Vertical="middle"/>
</UIMenu>
```

Listing 2: Extract from the Exploremenu.xml file

Note that, at this time, the inter-dialog navigation is *partially* automatic because dialog navigation, depending on application specific parameters, still has to be explicitly coded by the designer of the VE.

As shown in the examples, next to the description of the interface elements, both event generation and inter-dialog navigation are supported within the description language. As a result of this, the look and behavior of the user interface can be altered without editing or recompiling the application code.

Specifying 2D and 3D user interfaces

The 3D user interfaces, specified by means of VRIXML, and the classic 2D GUIs have several similarities. The most obvious correspondence is the look-and-feel of both interfaces, which is quite self-evident, since the user interface widgets are designed, based on their two- dimensional counterparts.
However, 3D interfaces contain several properties that are non-existent in two dimensions. The 3D position is one of the visible properties but lots of the additional properties are invisible. In our specific case, several of these properties concern haptics.
One of these haptic properties are tactile textures, which offer possibilities for the user to feel the structure of certain regions. Another example are force fields. Currently, each button contains a spherical magnetic force field at its centre in order to prevent the user from drifting, while operating the button.
The magnitude of this force field or the structure of the tactile texture are properties that can easily be integrated within VRIXML.

CONCLUSION

In this paper we presented a framework through which we try to abstract the description of virtual environments and the interaction within these computer generated worlds. In order to describe the user interface, which is part of those interaction techniques, we proposed a description language, called VRIXML. Through the use of a user interface description language, we created possibilities to design a user interface or adjust its behavior within a VE.
We are convinced that the use of this UIDL is a good first step in our VR-DeMo project.

FUTURE WORK

The main goal of our ongoing work is to develop an extensive graphical tool that allows easy creation of a user interface within the presented VE framework. Once the interface is designed, and all functionality is coupled to it, the tool will automatically generate the XML description(s) for the created user interface. In addition to the generation of the user interface description, the toolset will support also several automatic code generation possibilities that are based on high level descriptions of the environment's functionalities.
The IDs that are used during the automatic generation process, are all based on a shared XML file in which the designer has to define all program constants such as event IDs, IDs of user interface elements, ... (e.g. `<EventID id="14">SetDiffuseColor</EventID>`).
Within the tool, the user does not have to worry about the value of these identifiers. Instead, he can use the string values, attached to them.
Another objective will be to further explore the possibilities of adding extra data to the user interface description, in order to describe the runtime behavior of

certain system functionalities. Finally we are aiming for automatic layout management. Currently, the designer of the interface has to specify the position of each widget within its parent widget as shown in Listing 1. A possible approach to automatize this process is presented in [14]. They use 2D constraints to specify spatial relations between the user interface widgets. These constraints can possibly be extended to three dimensions.UI

ACKNOWLEDGEMENTS

The research at EDM is partly funded by the Flemish Government and EFRO (European Fund for Regional Development).
The VR-DeMo project (IWT 030284) is directly funded by the IWT, a Flemish subsidy organization.

REFERENCES

1. Marc Abrams and Constantinos Phanouriou. UIML: an XML language for building device-independent user interfaces. In *Proceedings of XML '99*, december 1999.
2. T. Anderson, A. Breckenridge, and G. Davidson. FBG: A graphical and haptic user interface for creating graphical, haptic user interfaces. In *Proceedings of the Fourth PHANToM Users Group Workshop*, Dedham, MA, USA, October 1999.
3. Farid BenHajji and Erik Dybner. *3D Graphical User Interfaces*. PhD thesis, Department of Computer and Systems Sciences Stockholm University and The Royal Institute of Technology, October 1999.
4. Joan De Boeck, Chris Raymaekers, and Karin Coninx. Aspects of haptic feedback in a multi-modal interface for object modeling. *Virtual Reality Journal*, 6:257–270, 2003.
5. Karin Coninx, Frank Van Reeth, and Eddy Flerackers. A hybrid 2D/3D user interface for immersive object modeling. In *Computer Graphics International '97*, pages 47–55, Diepenbeek, BE, January 1997. IEEE Computer Society Press.
6. Joan De Boeck, Chris Raymaekers, Erwin Cuppens, Tom De Weyer, and Karin Coninx. Task-based abstraction of haptic and multisensory applications. accepted for Eurohaptics 2004, Munchen, DE, June 5–7 2004.
7. M. Deering. The holosketch VR sketching system. *Communications of the ACM*, 39(5):54–61, May 1996.
8. Inc eNode. World Wide Web, `http://www.enode.com/`, 2002.
9. Voice XML Forum. VoiceXML: Voice extensible markup language, March 2000.
10. WAP Forum. Wireless application protocol white paper, June 2000.
11. M. Green and S. Halliday. A geometric modeling and animation system for virtual reality. *Communications of the ACM*, 39(5):46–53, May 1996.
12. Jed Hartman and Josie Wernecke. *The VRML 2.0 Handbook*. Silicon Graphics Inc, 1996.
13. Kris Luyten and Karin Coninx. An xml-based runtime user interface description language for mobile computing devices. In Chris Johnson, editor, *Interactive Systems: Design, Specification, and Verification*, volume 2220, pages 1–15, Glasgow, Scotland, UK, June 13–15 2001. 8th International Workshop, DSV-IS 2001, Springer-Verlag.
14. Kris Luyten, Bert Creemers, and Karin Coninx. Multi-device layout management for mobile computing devices. Technical Report TR-LUC-EDM-0301, Limburgs Universitair Centrum, Universitaire Campus gebouw D, B-3590 Diepenbeek, September 2003.
15. Thomas H. Massie and Kenneth J. Salisbury. The PHANToM haptic interface: A device for probing virtual objects. In *Proceedings of the ASME Winter Annual Meeting, Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems*, Chicago, IL, November 1994.
16. Chris Raymaekers and Karin Coninx. Menu interactions in a desktop haptic environment. In *Proceedings of Eurohaptics 2001*, pages 49–53, Birmingham, UK, July 1–4 2001.
17. Chris Raymaekers, Tom De Weyer, Karin Coninx, Frank Van Reeth, and Eddy Flerackers. ICOME: An immersive collaborative 3d object modeling environment. *Virtual Reality Journal*, pages 265–274, 1999.
18. Lawrence J. Rosenblum and Michael R. Macedonia. PHANToM-based haptic interaction with virtual objects. *IEEEComputer Graphics and Applications*, 17(5):6–10, September-October 1997.
19. Kenneth J. Salisbury. Making graphics physically tangible. *Communications of the ACM*, 42(8):75–81, August 1999.
20. Ben Shneiderman. *Designing the User Interface - Effective Strategies for Effective Human-Computer Interaction*, chapter 7: Menu Selection, Form Fillin, and Dialog Boxes, pages 235–274. Addison-Wesley, third edition, 1998.

Multimodal Dialog Description for Mobile Devices

Steffen Bleul
Paderborn University / C-LAB
Fuerstenallee 11,
Paderborn, Germany
bleul@upb.de

Wolfgang Mueller
Paderborn University / C-LAB
Fuerstenallee 11,
Paderborn, Germany
wolfgang@c-lab.de

Robbie Schaefer
Paderborn University / C-LAB
Fuerstenallee 11,
Paderborn, Germany
robbie@c-lab.de

ABSTRACT
The provision of personalized user interfaces for mobile devices is a challenging task since different devices with varying capabilities and interaction modalities have to be supported. Multiple variants of different UIs for one application almost enforces the employment of a model-based approach in order to design one interface and to adapt to or render it on those devices. This position paper presents a new dialog modelling language named DISL (Dialog and Interface Specification Language) that is based on UIML and DSN (Dialog Specification Notation). DISL supports the modelling of advanced dialogs in a comprehensive way. The dialog descriptions are device- and modality-agnostic and therefore highly scalable with focus on limited devices, like mobile phones.

1 INTRODUCTION

With the wide ability of considerably powerful mobile computing devices, the design of portable interactive User Interfaces (UIs) is posed to new challenges, as each device may have different capabilities and modalities for UI rendering. The growing variety of different mobile devices to access information on the Internet has induced the introduction of special purpose content presentation languages, like WML [17] and CompactHTML [8]. However, their application on limited devices is cumbersome and most often requires advanced skills. Therefore, we expect that advanced speech recognition and synthesis will soon complement current technologies for user-, hardware-, and situation-dependant multimodal interaction in the context of embedded and mobile devices. First applications are developed in the area of Ambient Intelligence (AmI) [1], which combines the areas multimodal user interface and ubiquitous/pervasive computing [18].

For generic multimodal user interface description languages, there are currently only very few activities. In the area of graphical user interface description languages, the User Interface Markup Language (UIML) [2] has been established and is currently available as UIML 3.0. UIML is mainly for the description of static user interfaces (structures) and their properties (styles) also leading to the description of User Interfaces, which are not completely independent from the target platform. The behavioural part of UIML is not well developed and does not give sufficient means to specify real interactive, state-oriented user interfaces. The same counts also for CUIML [13], which is a bit more flexible by introducing generic components that can be used for multimodal interaction. VoiceXML [9] is widely recognized as a standard for the specification of speech based dialogs. In addition to both, InkXML [16] has been defined to support interaction with hand writing interfaces. However, UIML, VoiceXML, and InkXML only cover their individual domains and do not integrate with other modalities. Beyond those, there are other XML-based multimedia languages for general interactive multimedia presentation, such as MHEG, HyTime, ZyX, and SMIL [3]. They enable simple authoring of rich multimedia presentations including layout, timing of streaming audio, video, images, text etc. as well as some very basic interactions in order to select a specific path in an interactive presentation. Considering all XML-based languages, only UIML and VoiceXML provide partial and SMIL limited support for user interaction description. Nevertheless, both are still rather limited for the specification of more complex state–based dialogs as they frequently appear in the interaction with mobile devices and remote control via those devices.

Numerous other approaches employ high level modeling techniques as e.g. task modeling like it is done in the TERESA project [10], or described in [4] and [6]. Our approach however concentrates on a lower level in order to define a dialog model which could also be generated from those higher level models. The description of the dialog and control model will be provided in [14] in more detail.

The W3C has established activities for an architecture for general multimodal interaction [7]. The Multimodal Interaction (MMI) Framework (cf. 1) defines an architecture for combined audio, speech, handwriting, and keyboard interaction as a set of properties (e.g., presentation parameters or input constraints); a set of methods (e.g., begin playback or recognition); and a set of events raised by the component (e.g., mouse clicks, speech events). The MMI framework covers

- multiple input modes such as audio, speech, handwriting,

and keyboarding;

- multiple output modes such as speech, text, graphics, audio files, and animation.

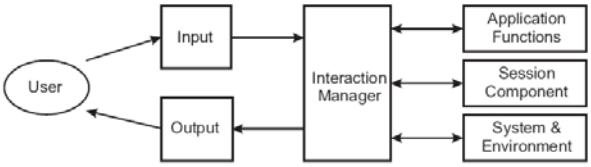


Figure 1. W3C Multimodal Interaction Framework

MMI concepts consider human user interaction via a so-called interaction manager. The human user enters input into the system, observes, and hears information presented by the system. The interaction manager is the logical component that coordinates data and manages execution flow from various input and output modalities. It maintains the interaction state and context of the application by responding to inputs from component interface objects and changes in the system and environment.

This paper introduces an instance of an MMI framework. We present the architecture of our architecture for the provision of multimodal UIs. In the context of that architecture, we introduce the XML-based Dialog and Interface Specification Language (DISL). DISL is based on an UIML subset, which is extended by rule-based descriptions of state-oriented dialogs for the specification of advanced multimodal interaction and the corresponding interfaces. DISL defines the state of the UI as a graph, where operations on UI elements perform state transitions. DISL's dialog part is based on DSN (Dialog Specification Notation), which was introduced to describe User Interface control models. Additionally, DISL gives means for a generic description of interactive user dialogs so that each dialog can be easily tailored to individual input/output device properties, e.g., graphical display or voice. In combination with DISL, we additionally introduce S-DISL (Sequential DISL). S-DISL is a sequentialized representation of DISL dedicated to the limited processing capabilities of mobile devices.

The remainder of this paper is structured as follows. The next section presents an architecture for multimodal UI provisioning. Section 3 introduces dialog modelling concepts and DISL. Section 4 gives the simple example of a remotely controlled media player before the paper closes with a conclusion and outlook.

2 ARCHITECTURE

Before going into the details of the modelling language DISL, we present a client-server architecture that provides user interface descriptions for mobile devices. This architecture allows controlling applications on the mobile device, on the server or using the device as a universal remote control as it is done in the pebbles project [11]. Having a UI server allows also a more flexible handling of UI descriptions as they can be transformed into specific target formats for mobile devices, which do not have dedicated DISL renderers. In fact, our DISL renderer for mobile phones also requires a pre transformation, which is done server side in order to establish a highly efficient parsing process on the client device. Figure 2 shows a simplified view of the architecture for use with mobile devices that are equipped with DISL (or more specifically S-DISL) renderers. For systems without DISL or S-DISL renderers, e.g., simple WAP-phones, the transformation component has to generate other target formats. However, in that case some of the advances by using DISL are lost.

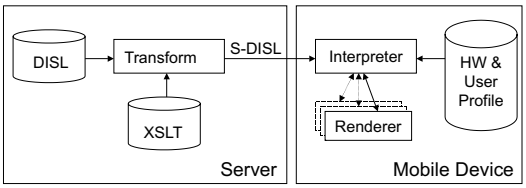


Figure 2. System Architecture

Since this architecture aims to support limited mobile devices with different interaction modalities, several constraints arise which influence the development of the dialog modelling language.

For supporting different modalities on a client device, the dialog representation, which is requested from the server, should be as generic as possible, so that a renderer can adapt it for a specific modality. The interpreter spans several renderers, one for each supported modality. In order to realize multimodal presentation, each generic widget is mapped to a concrete widget in the targeted modality. Interaction handling is performed, as each input event for a concrete widget in a specific modality is mapped reversely to a generic widget, so that the DISL control model can process the content which is detailed in [14].

Currently available mobile phones communicate over GSM networks, where network traffic produces costs to the user. Therefore the number of connections to the server and the amount of data transported should be limited, which means that processing and changing the dialog states has to be done on the mobile client.

We should also take into account that network connections are not reliable all the time. The UI should not freeze in case of errors or late server responses; therefore a concept of timed dialog state transitions is required.

As mobile phones usually come with low processing power and limited heap space, The dialog descriptions should be easy to parse which lead to the development of the S-DISL format, presented in Subsection 3.4.

3 DIALOG DESCRIPTION

For describing dialogs, UIML [2] is a good starting point as its meta interface model provides a clear separation between logic and presentation. The interface part of UIML sepa-

rates between structure, style, content, and behaviour. We have taken this interface modelling structure and extended the behavioural part with DSN [5] concepts. Additionally, in order to meet the requirement of supporting the most limited devices as well as different interaction modalities, we provide a vocabulary of generic widgets. The notion of generic widgets is inspired amongst others by [12] where a generic UIML vocabulary for the generation of graphical and voice user interfaces is defined.

3.1 Generic Vocabulary

We tried to find out the most basic elements, which are of importance for graphical UI, voice interaction gestures and other modalities and come up with following items that can be grouped into informative, interaction and collection elements.

As informative elements there are variablefield and textfield. The purpose of both informative elements is to provide feedback to the user. However, variablefield is designed to show the simple value or status of a variable, while textfield is for displaying or speaking larger portions of text, which means that a renderer has to supply additional means for navigation through larger information chunks, e.g., scrollbars for visual interfaces or interrupts in speech dialogs. These two elements obviously allow rendering for voice or graphical / text based dialogs, but even minimal output modalities are possible. For example, we can specify the variablefield to be an alert, which then could be rendered as beeps, vibrations or flashing lights.

For interaction purposes, the elements command, confirmation, variablebox and textbox are allowed. As variablefield and textfield are used for output of values and text, variablebox and textbox are used for input of the corresponding data. The difference between commands and confirmation lies in the user initiative. While the user can trigger a command, e.g., by pressing a button, the system may require confirmations when performing a specific task.

For structuring and selection of structured elements, choicegroup and widgetlist are provided. While the widgetlist just groups elements together according to the structure the modeller determines, the choicegroup is used to group elements from which one or more can be selected. The renderer is again responsible how the logical grouping is communicated to the user, e.g., by drawing boxes or in voice dialogs by prompting something as ”You have following choices: A, B, C...”.

For the case that we did not think of a basic widget, which is necessary for future interaction modalities, or to use platform specific code, we provide genericfield, genericcommand and genericbox as extension elements. They allow the use of arbitrary binary data.

Common to all Elements is that – provided they are used – they have to be attributed with several properties that specify them more clearly and by that provide hints to the renderer. We identified following property groups for our generic vocabulary:

- Render properties are used to describe the widgets and to guide the rendering process, for example by specifying labels.
- Render flags can be employed to determine if widgets have to be rendered or not. This is useful to cut widgets without modifying the interface structure.
- Interaction properties are needed to specify the value of an interaction object.
- Interaction flags show the current state of an interaction element, e.g. whether an interaction element is activated or if an element has been selected.
- dynamic properties are used for properties that are inherited for every element of a collection.
- System properties are provided by the system itself. For example for mobile phones, a system property could provide the number of characters that fit into a text line.

3.2 DISL Structure

DISL employs the same global structure as UIML but does not allow the peers section, because peers would destroy the concept of generality in our approach. By forcing not to use platform-specific widgets or logic, we can ensure that DISL descriptions can be rendered or easily transformed on most different devices and even for varying interaction modalities. Therefore, instead of using peers, we presume dedicated DISL renderers, which interpret generic UI elements or would otherwise perform a complete transformation of the DISL description to a target language. On the other hand, communication with the back-end application is still required and that is applied through the calls, which are executed in the action part of the behavior section.

Interfaces in the DISL language consist of structure, style, and behavior. The structure part in DISL is less complex than in UIML and consists of a set of nested generic widgets, as described above. The different types of widgets are instanced by attributes, which means that the set of possible widgets is fixed with the DTD. However, the set of properties for each widget is for the moment open and depends on which properties for each widget are supported by the renderer or transformation application. In our DISL specification we defined a set of properties, which is mandatory to achieve meaningful dialog modelling.

The widget properties are specified in the ”style” section of DISL. There within each ”part” element, the properties for the corresponding widgets from the ”structure” section are set, which follows the same type of separation from structure and style as in UIML.

3.3 Advanced Dialog Control

Major changes to UIML, apart from the definition of a fixed set of generic widgets, are in the behavioural section. As many approaches for specifying the dialog-flow are based on state transitions, the simpler modelling concepts can end up

in a difficult to handle large set of states. Therefore, we use concepts inherited from DSN [5], which is able to process sets of states during each transition and by that reducing the number of transition rules. Following example should make this concept clearer:

```
USER INPUT EVENTS
  switches (iVolumeUp, iVolumeDown,
            iPlay, iStop)

SYSTEM STATES
  volume (#loud  #normal #quiet)
  power      (#on #off)

RULES
  #normal #on iVolumeUp -->  #loud
```

It defines four interaction based events and two states. The rule fires when the interaction event iVolumeUp occurs, volume equals #normal, and power is #on. After firing, the rule sets volume to #loud.

This concept is reflected in the behaviour section, where the traditional UIML-based approach is extended with possibilities to specify variables, events, rules (operating different from UIML-rules) and transitions. Variables are used as content elements of the control model, which can be assigned to influence the dialog flow. For example a variable ”volume” could keep the current volume of a music application and will be set to zero, if within a dialog, a mute-control is triggered.

Based on these variables and events we can model powerful rules that modify the dialog state. In the simplest form rules are used to set a Boolean value, but normally they evaluate a complex condition that evaluate Boolean expressions over variable content, constants, numerous events like timeouts, results of external calls, periodic events and much more.

After having specified a set of rules, transitions are specified. These transitions implement the DSN-functionality as they allow the evaluation of several conditions at the same time. Only if all conditions are met, the transition may fire. Firing means that the action part of the transition is evaluated.

The action part allows calls to the backend application, restructuring the UI but also exchanging a complete interface, statements and loops, e.g., for assigning variables with new values. Statements are also used to activate self-defined events, while on the other hand several system events can occur e.g. when the external communication with the backend application is timed out.

This event mechanism introduces a new concept, which is derived from the concept of timed transitions in ODSN [15]. Events support advanced reactive UIs on remote clients, since they provide the basis for, e.g., timers. DISL events contain an action part as transitions. However, this action is not triggered by a set of rules evaluating to true rather it depends on a timer, which is set as an attribute. An event may be fired only once after the predefined timer expired or it may periodically fire. It is also possible to specify the activation or deactivation of events.

The following example shows, how the event mechanism is used to periodically check, which song is currently playing in a remote music player. Additionally, it outlines how external calls can be applied.

```
<event id="checkplaying" activated="yes"
       repeat="yes" timer="20s">
  <action>
    <call source="http://.../servlet"
         id="getsong" synchronized="yes"
         timeout="5s" maxsize="2">
      <parameter id="request">
        <value-of>getplaypos</value-of>
      </parameter>
    </call>
    ...
  </action>
</event>
```

A call consists of a source. This is typically an http request but any other protocols can be supported as well. The call represents the communication with the communication with the real application. The call id is used as a pointer to the return value of the application, which can also be an exception in case of an error. The timeout parameter is used to catch unexpected errors, e.g., when an application is not responding due to a network failure. Rules based on such unexpected errors can be specified, so it is up to the interface designer to model the behaviour after the timeout. The timer based event mechanism also allows client based synchronization with the backend application since querying external resources can modify internal UI-states.

The next example illustrates a DISL rule by specifying the volume control of a media player:

```
<behavior>
  <variable id="Volume" internal="no"
            type="integer">128</variable>
  <variable id="incVolumeValue" internal="no"
            type="integer">20</variable>
  ...
  <rule id="IncVolume">
    <condition>
      <equal>
        <property-content
             generic-widget="IncVolume"
             id="selected">
           yes
        </property-content>
      </equal>
    </condition>
  </rule>
  ...
  <transition>
    <if-true rule-id="IncVolume"/>
    <action>
      <statement assignment="add">
        <variable-content id="Volume"/>
        <variable-content id="incVolumeValue"/>
      </statement>
      <statement>
        <property-content id="visible"
```

```
                    generic-widget="Apply">
                    yes
          </property-content>
        </statement>
          ...
      </action>
    </transition>
<behavior>
```

First, variables for the current volume and a value for increasing the volume are assigned. The rule "IncVolume" implements the condition that evaluates to true, if the widget "IncVolume" is selected. After the conditions of each rule are evaluated we have to decide which transitions will be fired. This is done for every transition, where the condition of the if-true tag is true, then a set of statements is processed in the action part. There, the "incVolumeValue" is added to the previous set volume, and statements update the UI, e.g.,setting a "yes" and "cancel" control.

3.4 DISL for Limited Devices

Since DISL is designed for mobile devices with limited resources limited, we developed a serialized form of DISL that allows faster processing and a smaller memory footprint, namely S-DISL. The idea behind S-DISL is that an S-DISL interpreter just has to process a list of elements rather than complex tree structures. On the one hand this saves processing time, on the other hand gives a smaller footprint for the interpreter, which both saves resources required for UI rendering. To achieve a serialized form, a preprocessor implements a multi-pass XSLT transformation of the DISL file to S-DISL.

The first two passes are used to flatten the tree structure. To avoid information loss, new attributes providing links, like "nextoperation", "nextrule" etc. have to be introduced. Through that, the 42 elements of the SDML DTD can be reduced to 10 basic elements. For example, all action elements are reduced to one with a mode attribute defining the type.

The next transformation step sorts the ten element types into ten lists. Ids are replaced by references and empty attributes are deleted in order to get a lean serialized document. The final output is a stream of serialized elements. Although the stream is bigger than the original tree structure, the saved processing time outweights the disadvantage. The size of the stream however can be additionally reduced by using the binaryXML.

4 EXAMPLE

To demonstrate the working architecture for DISL, we give an example, which already is already completely implemented and in use. The idea is to control home entertainment equipment through mobile devices. More specifically, we control the playback of MP3 files on a PC by a J2ME-MIDP enabled mobile phone[1].

[1]In order to become attractive, consider cost-free, short-range Bluetooth communication of a mobile phone, so that it can be used as an universal remote control within the home environment. However, the current implementation applies bundled GSM transmission based communication (GPRS) with the server.

On a PC, a user is able to use a full fledged graphical user interface as it comes, e.g., with Winamp (see Fig. 3). However, that UI cannot be rendered on a mobile phone with a tiny display. Therefore, we have applied the aforementioned concepts in developing a generic user interface, which enables control of the MP3 player. This generic UI can be implemented as a service, which can be downloaded and used by the mobile phone.

Figure 3. GUI of Windows based MP3 Player

The generic UI - in DISL Notation - mainly describes the control model together with rendering hints. It is transformed in a very memory and space efficient manner to the intermediate S-DISL format through several XSLT transformation steps and finally transmitted to the mobile device, which runs the interpreter and renderer given as a Java Midlet.

The UI for our music player consists of controls to switch the player on or off, to start playback, to stop playback, to mute or to pause the sound, and to jump to the next or the previous title; volume control is also possible.

The collection of these controls is provided as a list of widget elements in the DISL description, which also describes the state transitions as well as their binding to commands of the backend application, i.e., the Winamp player. The following S-DISL code fragment gives the widget list for volume control:

```
<structure>
  <widget id="TitleScreen"
          generic-widget="variablefield"/>
  <widget id="ActVolume"
          generic-widget="variablefield"/>
  <widget id="SetVolume"
          generic-widget="variablebox"/>
  <widget id="IncVolume"
          generic-widget="command"/>
  <widget id="DecVolume"
          generic-widget="command"/>
  <widget id="Cancel" generic-widget="command"/>
  <widget id="Back" generic-widget="command"/>
  <widget id="Apply" generic-widget="command"/>
</structure>
```

The structural part of the interface description is followed by a style description for each supported widget. The style elements provide information for the renderer. For example, it defines whether the widget is visible or not. The following code fragment shows the style component for one widget:

```
<part generic-widget="IncVolume">
  <property id="title">Increase Volume</property>
```

```
  <property id="description">
      Increases Volume by 10
  </property>
  <property id="help">
    Every time this command is activated
    the volume will be increased by 10%
  </property>
  <property id="selected">no</property>
  <property id="visible">yes</property>
  <property id="activated">yes</property>
</part>
```

DISL structure and style specifications are quite similar to UIML. The following behavioural part largely differs from UIML and extends UIML towards state oriented DSN. The specification consists of rules and transitions as introduced before. We only show one transition illustrating the action of the "increase volume" command. The transition fires, after the "IncVolume" rule becomes true. Then, the value of the variable "IncVolumeValue" is added to the variable "Volume". The following actions then switch the "Apply" and "Cancel" widgets to visible[2].

```
<transition>
  <if-true rule-id="IncVolume"/>
  <action>
    <statement assignment="add">
      <variable id="Volume"/>
      <variable id="IncVolumeValue"/>
    </statement>
    <statement>
      <property id="visible"
                generic-widget="Apply">
        yes
      </property>
    </statement>
    <statement>
      <property id="visible"
                generic-widget="Cancel">
         yes
      </property>
    </statement>
  </action>
</transition>
```

Commands to the backend application are provided as http requests, which are handled by the Interaction Manager who is responsible for passing the commands to the application. The UI Interaction Manager can employ the functionality of a webserver, since WAP enabled phones and PDA's typically support HTTP. In our implementation, the communication part of our system is written as a set of servlets based on the Apache webserver. In our test environment, the player software to be triggered resides on the same machine as the Webserver, but this can be easily changed to a distributed system, e.g., with the OSGi Framework (http://www.osgi.org/). That would allow controlling applications on multiple target devices, for example, TV, VCR, radio.

The client we are currently using is a Siemens S55 mobile phone (see Fig. 4) that comes with Java MIDP which supports simple basic UI elements. The pictures showing some interfaces on the mobile phone where taken from an emulator, as photographs from the real device are not clear enough. When the music player application is selected, the UI is requested from the web server and all internal structures are initialised, before the UI can be rendered. This procedure has to be performed only once at the initial startup and may take some seconds. Afterwards even operations, which require server communication, are as fast a one can expect when communicating with a WAP server.

Figure 4. UI rendered on Mobile Phone

Figure 5: UI on Siemens M55: emulator (left) and mobile phone (right)

5 CONCLUSION

This paper introduced an multimodal UI provisioning architecture together with the XML-based Dialog and Interface Specification Language DISL. DISL is based on an extended UIML subset. The extensions are based on DSN (Dialog Specification Notation). Our current implementation has demonstrated the feasibility for mobile phones. Major parts of MIRS run on an Apache webserver in combination with a J2ME MIDP1.0 enabled Siemens M55 mobile phone. The implementation currently covers the complete definition of DISL, its transformation to S-DISL by a XSLT transformer, the complete S-DISL interpreter, as well as a graphical renderer.

Yet missing is advanced support for UI designers through modeling tools. For the moment, additional convenience can only be achieved by using standard XML editing tools.

In order to complete and test the current implementation we still have to extend it by a voice-based renderer and voice recognition. However, currently available mobile phones as well as PDAs do not provide sufficient processing power; neither for software-based real-time voice synthesis nor for

[2] "visible" is interpreted as "audible" for voice rendering

speech recognition. Therefore, we have established a PC-based test bed, which also is also used for the evaluation of user and hardware profile dependent rendering of multimedia information.

REFERENCES

1. E. Aarts. Ambient intelligence in homelab, 2002. Royal Philips Electronics.
2. M. Abrams, C. Phanouriou, A. L. Batongbacal, S. M. Williams, and J. E. Shuster. UIML: an appliance-independent xml user interface language. In *Computer Networks 31, Elsevier Science*, 1999.
3. S. Boll, W. Klas, and U. Wertermann. A comparison of multimedia document models concerning advanced requirements. Technical report, Computer Science Department, University of Ulm, Germany, 1999.
4. T. Clerckx, K. Luyten, and K. Coninx. Generating context-sensitive multiple device interfaces from design. In *Proceedings Fifth International Conference on Computer Aided Design of User Interfaces (CADUI 2004)*. Kluwer Academic, 2004.
5. M. B. Curry and A. F. Monk. Dialogue modelling of graphical user interfaces with a production system. In *Behaviour and Information Technology, Vol. 14, No. 1, pp 41-55*, 1995.
6. J. Eisenstein, J. Vanderdonckt, and A. Puerta. Applying model-based techniques to the development of uis for mobile computers. In *Proceedings of Intelligent User Interfaces Conference (IUI2001)*, 2001.
7. D. Raggett (eds.) J. A. Larson, T.V. Raman. W3c multimodal interaction framework, May 2003. W3C NOTE 06 May 2003.
8. T. Kamada. *Compact HTML for Small Information Appliances, W3CNote*, Februar 1998.
9. S. McGlashan et al. Voice extensible markup language (voicexml) version 2.0, w3c proposed recommendation, 2004. http://www.w3.org/TR/voicexml20.
10. G. Mori, F. Paternò, and C. Santoro. Tool support for designing nomadic applications. In *Proceedings of the 8th international conference on Intelligent user interfaces*, 2003.
11. J. Nichols, B. A. Myers, M. Higgins, J. Hughes, T. K. Harris, R. Rosenfeld, and M. Pignol. Generating remote control interfaces for complex appliances. In *CHI Letters: ACM Symposium on User Interface Software and Technology, UIST'02*, 2002.
12. J. Plomp and O. Mayora-Ibarra. A generic widget vocabulary for the generation of graphical and speech-driven user interfaces. *International Journal of Speech Technology*, 5(1):39–47, January 2002.
13. C. SAndor and T. Reicher. Cuiml: A language for generating multimodal human-computer interfaces. In *Proceedings of the European UIML Conference*, 2001.
14. R. Schaefer, S. Bleul, and W. Mueller. A novel dialog model for the design of multimodal user interfaces. In *Submitted for publication*, 2004.
15. G. Szwillus. Object oriented dialogue specification with odsn. In *Proceedings of Software-Ergonomie '93, Teubner, Stuttgart*, 1997.
16. Z. Trabelsi, S.-H. Cha, D. Desai, and Ch. Tappert. A voice and ink xml multimodal architecture for mobile e-commerce system. In *Proceedings of the second international workshop on Mobile commerce, 2002 , Atlanta, Georgia, USA*, 2002.
17. WAP Forum. *Wireless Markup Language Specification Version 1.1*, Juni 1999.
18. M. Weiser. The computer for the 21st century, 1991. Scientific American 265(3): 94-104.

Extending an XML environment definition language for spoken dialogue and web-based interfaces

Pablo A. Haya	Germán Montoro	Xavier Alamán	Rubén Cabello	Javier Martínez
EPS-UAM	EPS-UAM	EPS-UAM	EPS-UAM	EPS-UAM
Madrid, Spain	Madrid, Spain	Madrid, Spain	Madrid, Spain	Madrid, Spain
+34 91 497 22 67	+34 91 497 22 10	+34 91 497 22 50	+34 91 497 22 68	+34 91 497 22 54
Pablo.Haya @uam.es	German.Montoro @uam.es	Xavier.Alaman @uam.es	Ruben.Cabello @uam.es	Javier.Martinez @uam.es

ABSTRACT

In this work we describe how we employ XML-compliant languages to define an intelligent environment. This language represents the environment, its entities and their relationships. The XML environment definition is transformed in a middleware layer that provides interaction with the environment. Additionally, this XML definition language has been extended to support two different user interfaces. A spoken dialogue interface is created by means of specific linguistic information. GUI interaction information is converted in a web-based interface.

Keywords

Interface design, XML, UIDL, intelligent environments, spoken dialogues, web interfaces.

INTRODUCTION

Within the ubiquitous computing [13] research area it is necessary the study of the design of transparent user interfaces for the interaction with intelligent environments [4]. These interfaces provide new ways of interaction [14], adapt to the users and the environment and offer new challenges to interface designers [11].

Intelligent environment interfaces can range from a GUI mobile-interface (for instance a web-based interface, accessible from a computer or a PDA) to a higher-level interface (such as a spoken dialogue or a gesture-based interface).

Given the dynamic characteristics of intelligent environments, these interfaces have to be easily configurable and adaptable [7, 10] and have to provide standard methods of definition and configuration.

Bearing in mind these conditions we have developed an XML-compliant language that allows to define the characteristics of an intelligent environment. Furthermore, we have added interface information to the language, creating a user interface description language (UIDL) that permits to automatically create a web-based interface and a spoken dialogue interface based on the environment information.

Here we present the main ideas of our XML-based language that defines the intelligent environment and these two interfaces. Next sections are organized as follows: first, we give brief overview of the user interface definition languages; next, we describe the environment representation through our XML language; after that, we present the definition of the web-based and spoken dialogue interfaces; next we explain the implemented environment and, finally, we present the conclusions.

USER INTERFACE DEFINITION LANGUAGES

XML stands as a solution for the standardization of the interoperability between applications. Therefore, new XML-compliant languages are employed to define user interfaces. These are the XML-compliant user interface definition languages (XML-UIDL). They have the advantage of being transparent to different interface technologies and providing a homogeneous resource for heterogeneous ways of interaction [1].

According to [12] these XML languages for interface representation must be applicable to any target, any delivery context, personalizable, flexible and extensible. On the other hand, they should separate the interface elements from their presentation. The user interface elements must be explicitly represented and in a format that can be rendered in any delivered context. The presentation information should be provided in an abstract form that is target and delivery-context independent.

Two representative languages are:

- UIML [2], an XML-compliant language which permits the creation of user interfaces for any device, any target language and any operating system. It describes the appearance of the interface, the user interaction with the interface and how it is connected to the application logic.
- XIML [9], an XML-based "interface representation language for universal support of functionality across the entire lifecycle of a user interface: design, development, operation, management, organization, and evaluation".

Other languages are XUL [6], that allows to build easily customizable graphical user interfaces for multiple

platforms. AAIML [15], an XML-based language used to communicate an abstract user interface definition for a service or device to a user's personal device. And XAML [8], the Microsoft XML based language employed for visual interfaces to define a layout of text, images and controls.

XML ENVIRONMENT DEFINITION

The physical environment is represented in a document, where each environment entity is described using an XML format. Entities are not only formed by the physical devices presented in the environment, but also by software applications, people definition or abstract concepts.

This XML representation also allows to describe the relationships between the environment entities. These relationships define the distribution of the environment (buildings, rooms, etc.), aggregations of people (by workgroups, range, etc.) and dynamic links between entities (the favorite paintings of a person, the output speaker for a music source, etc.).

The XML information is processed by a parser and transformed in a middleware layer, which will act as an interaction layer between the user interfaces and the physical environment.

The middleware implementation lies on a global data structure, called blackboard [5]. This blackboard is a model of the world, where all the prominent information related to the environment is stored. The blackboard provides an asynchronous communication mechanism. Senders publish environment information in the blackboard, and receivers can be subscribed to these changes or pull them directly from the blackboard. This mechanism permits a loosely-coupled interaction among senders and receivers given that it is not needed that either both of them are active at the same time or they know each other. Therefore, the blackboard allows to communicate environment changes, finding available devices and revealing if a device has been added or removed.

Environment representation

The environment information stored in this blackboard can be viewed as a two-layered structure. On the one hand, a relationship layer has information about the relations between entities. On the other hand, an entity layer stores information about each particular entity.

The relationship layer is a non-directed graph where each node is an entity. Each entity node represents relevant environment information such as physical devices, software applications, occupants or abstract concepts. Arcs between entity nodes denote some kind of relationship (composition, aggregation, association, etc.). For example, the location of a person is modelled as an arc between that person and the room where s/he is located. Given that we employ non-directed arcs, reciprocal relations are also modelled. Therefore, each room has a relationship with every one of its occupants.

Every entity node has assigned a name. This is a unique alphanumerical string. This way, the node name univocally represents the entity. Moreover, entity nodes hold extra information that indicates the entity type (an entity can be a device, a person, a room, etc.).

The entity layer is composed as follows. Every entity has a collection of properties. Entities of the same type inherit a set of common properties, which defines their specific characteristics. Besides, the entities can define new common properties, called parameters, which represent custom information for the entity.

The composition of each environment entity is reflected in the blackboard as a tree structure. The tree root is one of the nodes of the previously described relationship graph. This node has a set of child nodes that defines its properties and parameters.

A property node constitutes an intrinsic and universally accepted feature of the entity. Properties have a name and a value. Thus, two properties that belong to the same entity must have distinct names. Values are leaf nodes that store literal values which can be of type string, integer or real. Besides, the changes on the values of the properties that represent physical variables are reflected in the real world. Thus, when an application or an interface needs to get or to change the physical state of a device, it only has to access to the right node in the graph and get or change its value.

Parameter nodes represent a set of specific features defined by an application or a group of applications, and allow to customize the entity model. Parameters hang of a parameter set node (aka *paramSet* node). Each group of applications can define its own *paramSet* independently of the rest. Parameters, like properties, are name-value pairs. Nevertheless, they can be associated not only to an entity but also to a property. This mechanism provides fine-grain parameterisation.

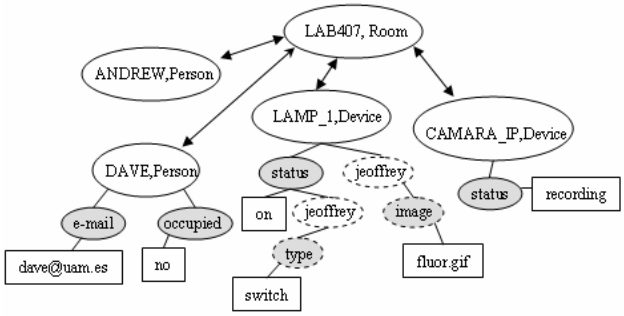


Figure 1. Blackboard: entities and their relationships

So, combining these two layers, the resulting blackboard structure can be seen like a graph of entities, where each entity is described as tree of properties and parameters. Figure 1 depicts a schematic blackboard graph. It contains five entity nodes, four property nodes and two *paramSet* nodes with one parameter each. Entities are within a blank circle, with their name and type (for instance, Andrew and person). Double-arrow lines indicate a bidirectional

relationship. Shadowed solid circles represent property nodes (for instance, e-mail), blank dashed circles represent *paramSets* (for instance, Jeoffrey) and shadowed dashed circles represent parameters (for instance, image). Finally, rectangles hold the node value (for instance, dave@uam.es).

This structure allows to organize the environment information using several abstraction levels. The deepest nodes represent more concrete properties, while the upper nodes in the hierarchy reflect structural relationships among entities.

Name Space

An entity node can be indexed by its name. Besides, a node can be located, starting from any entity node and following the relationship path. This is called the node path. It is composed by a list of tokens separated by the slash character. Their order is determined as follows: the first token of the path is the word "name", the second one must be the entity name and the next tokens come as the result of concatenating the names of all the intermediate nodes until the target node. For instance, in the example showed in the figure 1, the lamp_1 status path is */name/lamp_1/status*.

In addition, wildcards can be used to substitute one or several tokens. This allows referencing several nodes at the same time. For example, based on the figure 1, */name/dave/** references all the properties, *paramSets* and related entities of the entity *Dave*. As a result it gets the following list: the e-mail and busy property nodes and the Lab_407 entity node.

Another two naming mechanisms are provided to improve the use of wildcards:

- Predefined hierarchy. This mechanism restricts the nodes that compose a path. It specifies how to go through the graph. To do this, each hierarchy defines a sequence of types of entities. For example, the first type of entity must be a room, the second one a device, etc… Therefore, when a wildcard is used, only the nodes that match with the expected type will be substituted. These hierarchies are called predefined because they are hard-wired. Following with the example of the figure 1, the path */roomdevice/lab407/*/props/status* is interpreted as follows: the initial token identifies the hierarchy *roomdevice*. This hierarchy establishes that the first type of entity must be a room followed by a device. The other nodes remain unrestricted. Therefore, this path references the value of the status of all the devices located in lab407.
- Typed hierarchy. This is a particular case of the previous mechanism. By default, there will be as many hierarchies as types of entities. The initial token of these hierarchies is the type of entity. For example, in the figure 1 there are three default hierarchies: person, room and device, so that */person/*/mail* retrieves the e-mails from everybody.

```
<entity name="id" type="type">
    <property name="name">value</property>
    <property name="name">value
      <paramSet name="name">
        <param name="name">value</param>
        <param name="name">value</param>
          .....
      </paramSet>
    </property>
    .....
    <paramSet name="name">
      <param name="name">value</param>
      <param name="name">value</param>
       .....
    </paramSet>
     <paramSet name="name">
      <param name="name">value</param>
      <param name="name">value</param>
       .....
    </paramSet>
    ......
    <entity name="name"/>
    …..
</entity>
```

Figure 2. XML template for an entity

Interaction with the blackboard

Interfaces do not interact directly with the environment physical entities but they only have access to the middleware information. So, the implementation details of an entity are hidden to the applications and these only have to use the same standard communication rules for any entity of the environment.

The middleware provides a set of operations that allows to retrieve the information stored in blackboard, make changes on the values of the properties and add or remove an entity or a relationship. To access or change the blackboard information, applications and interfaces employ a simple communication mechanism through the HTTP protocol, by means of XML-compliant messages.

Figure 2 shows an XML representation of a generic entity obtained from the blackboard.

Thereby, the initial backboard structure can be generated from a set of XML files that store the environment configuration.

As we have seen in this section it is simple and standard to describe the environment, retrieve the state of its entities or

change it. The XML-compliant definition language serves as a standard tool to specify the characteristics of the environment. Once created, to get or change the physical state of an entity of the environment or add or remove new entities is also possible by means of standard instructions.

XML INTERFACE DEFINITION

Besides the definition of the entity properties, employed to build the middleware layer, the entities have associated other XML information employed to automatically build diverse user interfaces.

Currently, our XML-compliant environment definition language supports the automatic construction of two different user interfaces: a spoken dialogue interface and a web-based interface.

Spoken dialogue interface

Spoken interaction becomes necessary for an intuitive communication between users and intelligent environments [3]. Considering this, we have added new XML dialogue tags to the environment description, in order to support the automatic creation of a Spanish dialogue interface.

Dialogues are associated to each entity, so that when a new entity appears in the environment a new dialogue allows the users to interact with that entity. If the entity is not part of the environment, the dialogue will not be available.

Each dialogue entity depends on the type of entity, so the entities of the same type will inherit the same kind of possible interactions. Entity dialogues can be customized for each entity, in order to distinguish between them. A supervisor is in charge of managing the dialogue interactions, resolving conflicts, for instance, when there are several entities of the same type, among many others.

```
<class name="fluorescent">
<property name="Status">
<paramSet name="dialogue">
 <paramSet name="sentence">
  <param name="verbPart">turn_on switch_on</param>
  <param name="objectPart">light</param>
  <param name="modifierPart"> </param>
  <param name="locationPart">ceiling above</param>
  <param name="indirectObjectPart"></param>
 </paramSet>
 <paramSet name="sentence" >
  <param name="verbPart">turn_on </param>
  <param name="objectPart">fluorescent</param>
  <param name="modifierPart"> </param>
  <param name="locationPart"> </param>
  <param name="indirectObjectPart"></param>
 </paramSet>
</paramSet>
</property>
</class>
```

Figure 3. Linguistic information for an entity definition

Each entity must have associated all the possible ways a user can interact with it. For this we have defined an initial set of linguistic parts, which tries to cover the possible interactions between the user and the entity. This set is formed by:

- A verb part, which corresponds with the action that the user wants to perform with the entity.
- An object part, related with the name that the user gives to the entity.
- An indirect object part, the person who receives the action.
- A modifier part, the kind of object part entity.
- A location part, which informs of the location of the entity in the environment.

The last two parts permit to distinguish between several entities of the same type. These linguistics parts allow the use of synonyms and there can be as many sets of parts as necessary for each entity. The figure 3 shows the definition of two different sets of linguistic parts for one entity of type fluorescent. Translating the case from Spanish, it is considered that a user could utter sentences of the type: "please, could you switch on the ceiling light" but not of the type "please, could you switch on the fluorescent" (for fluorescents, users only employ the verb turn on). Besides, some parts contain synonyms (turn on and switch on, or ceiling and above).

```
<entity name="Lamp_1" type="fluorescent">
 <property name="status">
  <paramSet name="dialogue">
   <paramSet name="sentence" >
    <param name="modifierPart">main</param>
   </paramSet>
  </paramSet>
 <property>
</entity>
```

Figure 4. Customized entity instance

To create an entity based on a defined type it is only necessary to declare an instance of the entity type. This entity instance inherits all the entity type definition properties, including the linguistic information. In many cases, it will not be necessary to customize this linguistic information, and to declare the entity will be enough to automatically add its dialogue interactions to the interface.

In other cases, the entity instance can be customized to adapt to the environment specific characteristics or distinguish itself from other entities of the same type. Figure 4 shows an entity instance customized for a specific environment.

Additionally, the entity type definition also has to declare:

- A grammar template, which serves as the skeleton to define the recognition grammar.

A grammar template has a set of common rules and empty linguistic parts (marked as nil). The nil marks can be filled in with the linguistic parts provided by the entity definition. Figure 5 shows a simplified section of an action grammar template for an imperative sentence. Besides these imperative sentences, it also supports noun sentences,

```
<imperative sentence> = <imperative verb> [<noun>];
<imperative verb> =   <imperative informal verb>
                    | <imperative formal verb>
                    | <infinitive verb>;
<imperative informal verb> = nil;
<imperative formal verb> = nil;
<infinitive verb> = nil;
```

Figure 5. Section of a grammar template

subjunctive sentences (in present, past, singular and plural) and interrogative sentences.

Every word in the set of linguistic parts is sent to a morphological analyzer. This gets its part of speech information and, based on it, retrieves its different forms. Then it adds each word form to the right grammar rule. For instance, based on the exampled showed in the figure 3, the morphological analyzer gets that *turn on* is verb, so that it gets all the possible declinations for that verb (in Spanish, verb declinations change for each mode, tense, number and person). Then, it adds the right forms to the rules <*imperative informal verb*> and <*imperative formal verb*>, among others.

This process is repeated with each linguistic part of an entity type, taking into consideration if the word is a noun, a verb, an adjective, etc.

Grammar templates employ fixed rules that not only combine the added words in a proper way but also allow to employ more general and natural utterances, avoiding to use commands. These sentences try to cover the whole corpus of possibilities that a person employs to address to the entity.

The entity designer can use any of the available grammar templates. A designer can employ the preexisting grammar templates or declare new ones. In this case, s/he only needs to keep the name of the rules that will be filled in with the entity linguistic parts, this is, rules of the kind <infinitive verb>, <singular male noun>, etc. S/he only has to declare the rules that correspond with linguistic parts that are necessary for the interaction, avoiding to declare those not needed.

And finally, it is necessary that the entity definition declares a pointer to two different methods:

- An action method, which receives the action requested by the user (the verb part) and performs that action with the entity. To do this task it serves of the middleware layer.
- A state method, which also receives the verb part and returns if the current state of the entity is the same or different to the user requested state. Again, it also serves of the middleware layer.

The action method is employed to execute the environment physical action requested by the user. It only has to be implemented once by the designer of entity type so the entity instances will automatically inherit this method.

The state method is utilized in the interaction process to determinate if the entity instance has to be processed. In the case that the entity has the same state as the requested by the user the dialogue interaction does not need to consider that entity and can continue processing other entities with a different state. Again, this method only has to be defined once by the designer of the entity type. The interface definition process will automatically inherit this method for every entity of the same type.

Both methods employ the middleware layer to communicate with the physical environment. To do this, they only have to specify the entity property that they want to interact with, if they want to get or set a value for this property and, in the last case, the value that they want to set. As it was explained above, this communication follows a standard process through the HTTP protocol.

Web-based interface

We have also developed a web based interface to control environment's devices and appliances. This interface is called Jeoffrey. It is a custom and partial view of the environment information stored in the blackboard.

The blackboard contains generic information regarding the number of rooms and the entities that it hosts. Each entity is represented in the blackboard. Its representation includes the properties required to interact with it. Additionally, new specific information has been added in order to create the Jeoffrey interface. It is composed by three parts hierarchically structured:

- The top level is a stand-alone list box containing the rooms of the environment. When the user selects a room, a new window will pop up.
- This new window shows a map of the room, which includes the location of the furniture and entities. The map layout is composed overlapping a fixed background image with the device representation images. Every time

the interface is loaded, the map is dynamically generated using the blackboard information.

- Finally, a custom control panel is showed when a user clicks on an entity, allowing to interact with it.

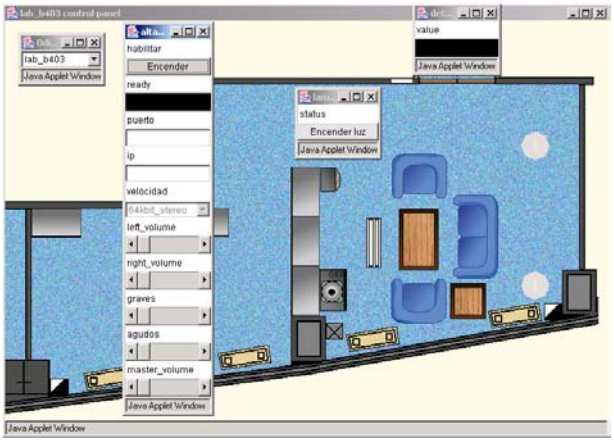


Figure 6. Jeoffrey's user interface.

Figure 6 shows a Jeoffrey user interface screenshot. The most left window is the root list box. The background window corresponds to the map that appears when a room is selected. Finally, the other three windows correspond to invoked entity control panels.

Jeoffrey gathers the information stored in the blackboard to dynamically render the user interface. The blackboard graph includes an entity node for each room and for each entity. A relationship between a room and an entity reflects that the entity is located in that room. This way, Jeoffrey can easily ask for all the rooms and, for each of them, which entities are inside.

Each entity includes several Jeoffrey's parameters that help to render its graphical interface. Figure 7 illustrates the Jeoffrey interface information of a fluorescent XML instance. Bold font is used to highlight the Jeoffrey's parameters. There are two *paramSets*. The first one is associated to the entity and contains three parameters. The image parameter defines its corresponding image file. The x and y parameters are the coordinates where this image will be drawn. The second *paramSet* is associated to the status property and defines its related widget.

As we have mentioned above, the interaction with the entities is managed by a custom control panel composed of widgets. This panel is customized depending on the entity properties. Each property is rendered into a widget that allows interacting with the entity property. There are five different generic widgets: text areas, switches, sliders, list boxes and alarms. Text areas permit to change the value of a string. Switches act as a toggle button associated to on-off properties. Sliders correspond to properties that take a value from an interval. List boxes define a list of possible values where the user can choose one. And finally, alarms are colored labels that change their color depending on the value of the property.

```
<entity name="Lamp_1" type="fluorescent">
 <property name="Status“>
  <paramSet name="jeoffrey”>
   <param name="type">switch</param>
   <param name="text_off">Turn on</param>
   <param name="text_on">Turn off</param>
   <param name="cmd_on">0</param>
   <param name=”cmd_off”>1</param>
   <param name="color_on">0x00FF00</param>
  </paramSet>
 </property>
 <paramSet name="jeoffrey">
  <param name="image">reflectante.gif</param>
  <param name="x">460</param>
  <param name="y">247</param>
 </paramSet>
</entity>
```

Figure 7. XML entity representation

As figure 7 shows, the fluorescent called Lamp_1 has only a status property. This property is associated with a switch widget. Besides, several switch parameters defining presentation features are established. These features are:

- The button text: this text changes depending on the state of the property. The "*text_off*" parameter is displayed when the light is off whereas the "*text_on*" parameter is showed when the light is on.
- The button color: by default the color is gray when the light is off. When the light is on, the color is defined by the "*color_on*" parameter.

Figure 8 illustrates the rendered control panel for a florescent and the image painted on the map.

Finally, the "*cmd_off*" and "*cmd_on*" parameters define the value of the status property that will be set when the button is pressed.

Figure 8. User interface for a fluorescent

When a user clicks on the picture of an entity, Jeoffrey reads the descriptions of its properties from the blackboard, translates the properties to widgets and generates a custom control panel. If the entity has more than one property, the

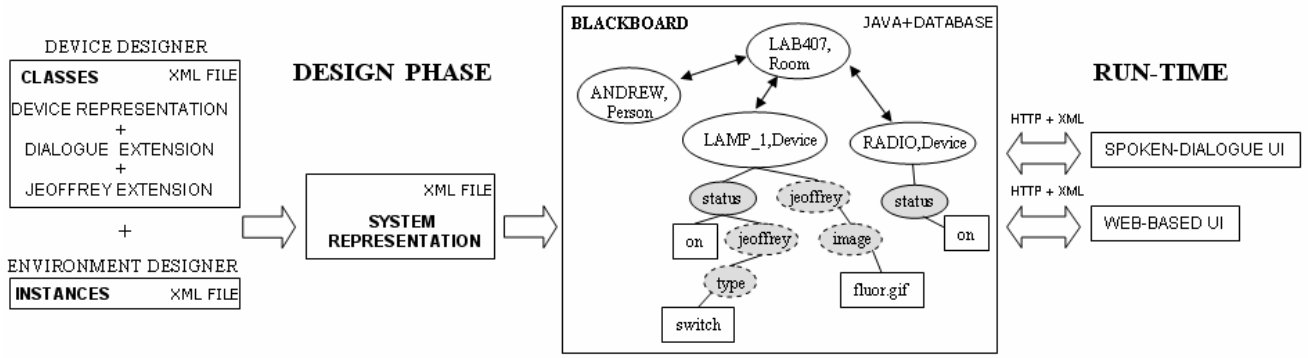


Figure 9. Overview of the system

control panel will be composed by the aggregation of the widgets corresponding to each property.

Jeoffrey employs the blackboard as a proxy to interact with the physical entities, for instance, to change the speaker volume, switch on the lights, etc., and to receive the changes occurred in the environment. Jeoffrey is subscribed to every event. All the changes in the state of an entity are reflected in the user interface. For instance, if a property has associated a widget alarm, when its value changes, the blackboard will notify this to Jeoffrey and it will modify the color of the alarm widget.

IMPLEMENTED ENVIRONMENT

Currently we have implemented a real intelligent environment that allows to control and interact with a fluorescent light, two reading lights, two dimmable lights, the main gate lock mechanism and an FM tuner with thirteen different radio stations, among other functionalities (such as sending messages, showing personalized paintings, etc.). All these devices are part of the laboratory number 407, so they hang from the path */lab407/device/*

These devices already come with their common XML entity definition, their action and state method and their associated grammar template. The device manufacturer is in charge of providing this information, so an environment designer only has to declare the instances of the elements and their distribution in the environment.

The environment designer declares the entities according to the template showed in the figure 2. If it is not necessary to customize the linguistic or web-based information for the current environment s/he will not have to declare any specific information related to the interfaces. Then the system adds the interface class information to the entity declaration. All this information is compiled to create the system representation file. With this file the system builds the blackboard, containing information about the environment and the interfaces (see an overview of the system in the figure 9).

For instance, in our developed environment the environment designer only has to declare the seven device members of the laboratory 407. Given that there are some devices of the same type, s/he will have to customize some interface information. In this case s/he employs new linguistic information to distinguish between the two dimmable lights, adding to the location part the words *left* and *right* respectively (see another example in figure 4). A similar case occurs with the two reading lights. Finally, the entity declaration is customized by specifying the coordinates x and y for the Jeoffrey's web-based interface.

With this, the system automatically creates the blackboard. Whenever the web-based interface is executed it consults the blackboard to create the interface showed in the figure 6. The information for all the devices of the laboratory 407 is retrieved employing the following path */roomdevice/lab407/*/*/jeoffrey/**. The dialogue information is retrieved in a similar way and it forms a linguistic tree as the core of the spoken interface.

Once the interfaces are created users can interact with the environment. The spoken dialogue interface allows natural interaction with the elements of the environment. Users can refer in different ways to the actions that can be taken with the devices and the system responds either uttering answers or executing actions. The interface supports interpretation of user sentences, based on the current physical context stored on the blackboard. A clarification request is produced when it does not have enough information to carry on an action. Besides, it supports anaphora resolution.

The dialogue interaction adapts to the elements of the environment and their state. Answers and system actions vary depending on the elements declared for each environment. This is done by means of the linguistic tree obtained from the blackboard at startup. This tree contains all the possible interactions with the environment and the entities that support them. Again, this is an automatic process and the environment designer only needs to declare the entities that form the environment.

A real example of an interaction produced in this environment is showed in the figure 10. It demonstrates how the system changes the interpretation of the same sentence for different states of the environment, interprets incomplete sentences or reacts when there are several entities of the same type.

User: Please, could you turn on the light?

System: What light would you like to turn on?

U: The reading light, please.

S: The one on the left or on the right.

U: The left light.

(The system turns on the left reading light)

U: Turn on the radio.

S: What station do you prefer?

U: I would like M80.

(The system turns on the radio with M80)

U: Please, turn it up.

(The system turns up the radio volume)

U: More.

(The system turns it up again)

U: I would like you to switch off…

S: Do you prefer to switch off the left reading light or the radio.

U: The radio, please

(The system turns off the radio)

U: I would like you to switch off…

(The system directly turns off the left reading light)

Figure 10. Spoken interaction with the environment

CONCLUSIONS

We have presented a graph model that allows to represent the entities of an intelligent environment and their relationships. This model is created using an XML-compliant language, and it is stored in a global data structure, called blackboard. A blackboard middleware provides a set of operations to interact with the graph model. An application can add or remove entities, retrieve or modify their state, and subscribe to the changes done by other applications.

Two user interfaces have been developed to interact with the environment. These interfaces are created by means of an extension of the environment XML model. The first extended language automatically creates a customized spoken dialogue interface. This language adds linguistic information to the XML model. The second one dynamically builds a web based interface. Again, new XML tags allow to specify GUI information.

The middleware and the interfaces have been developed in a real environment. It is composed of several devices, including different types of lights, sensors, a door opening mechanism, an FM tuner, etc. Both interfaces provide real interaction with these devices.

ACKNOWLEDGMENTS

This work has been sponsored by the Spanish Ministry of Science and Technology, project number TIC2000-0464.

REFERENCES

1. Abrams, M., Phanouriou, C., Batongbacal, A.L., Williams, S., and Shuster, J.E. UIML: An Appliance-Independent XML User Interface Language. In *Proceedings of the Eighth International WWW Conference*, Toronto, Canada, 1999.
2. Ali, M.A., Pérez-Quiñones, M.A., Abrams, M., and Shell, E. Building Multi-Platform User Interfaces with UIML. In *Proceedings of CADUI*, 2002.
3. Brumitt, B., and Cadiz, JJ. "Let there be light!" Comparing interfaces for homes of the future. In *Proceedings of INTERACT '01*, 375–382, 2001.
4. Coen, M.H. Design Principles for Intelligent Environments. In *Proceedings of the AAAI Spring Symposium on Intelligent Environments*, Palo Alto, California, 1998.
5. Engelmore, R., and Morgan, T. Blackboard Systems. *Addison-Wesley*, 1988
6. McFarlane, N. Rapid Application Development with Mozilla. Bruce Perens' Open Source Series. *Prentice Hall*, 2003
7. Paternò, F., and Santoro, C. One Model, Many Interfaces. In *Proceedings of CADUI*, 2002.
8. Petzold, C. Create Real Apps Using New Code and Markup Model. *MSDN Magazine*, January 2004.
9. Puerta, A. and Eisenstein, J. XIML: A Universal Language for User Interfaces. *White paper*. Available at http://www.ximl.org/Docs.asp. 2001.
10. Rayner, M., Lewin, I., Gorrell, G., and Boye, J. Plug and Play Speech Understanding. *2nd SIGdial Workshop on Discourse and Dialogue*, September 2001.
11. Shafer, S., Brumitt, B., and Cadiz, JJ. Interaction Issues in Context-Aware Intelligent Environments. *Human-Computer Interaction*, 16, 363-378, 2001.
12. Trewin, S., Zimmermann, G., and Vanderheiden, G. Abstract user interface representations: How well do they support universal access?. In *Proceedings of the 2nd ACM International Conference on Universal Usability*, Vancouver, Canada, 2003.
13. Weiser, M. The computer of the 21st century. *Scientific American*, 265, 3, 66-75, 1991.
14. Weiser, M. The world is not a desktop. *ACM Interactions*, 1, 1, 7-8, 1994.
15. Zimmermann, G., Vanderheiden, G., and Gilman, A. Universal Remote Console Prototyping of an Emerging XML Based Alternate User Interface Access Standard. In *Proceedings of the Eleventh International WWW Conference*, Hawaii, 2002.

IM^{2}L: A User Interface Description Language Supporting Electronic Annotation

Daniela Fogli
Dipartimento di Elettronica per l'Automazione
Università di Brescia, Italy
fogli@ing.unibs.it

Giuseppe Fresta
Istituto di Scienza e Tecnologia dell'Informazione (ISTI)
CNR, Pisa, Italy
giuseppe.fresta@isti.cnr.it

Andrea Marcante, Piero Mussio
Dipartimento di Scienze dell'Informazione
Università di Milano, Italy
{marcante,mussio}@dico.unimi.it

ABSTRACT
The user interface description language IM^{2}L (Interaction Multimodal Markup Language) - an XML compliant language - is introduced and its interpreter is discussed. The design of IM^{2}L is motivated by the need to support the activity of electronic document creation, management and updating in scientific and technical fields. In these domains, annotation emerged as the basic operator for electronic document management. IM^{2}L is a user interface description language in that an IM^{2}L program defines an interactive environment – including its interface. An IM^{2}L program can be interpreted by an XML processor and physically made active by an adequate application.

Keywords
XML, user interfaces, electronic document, electronic annotation.

1. INTRODUCTION

Electronic documents (e-documents) appear as a new media, complementing the traditional documents in recording, evolving and making available community knowledge. In the electronic world, documents become 'electronic', in that they are no more recorded on a permanent support, but exist 'virtually' as the results of the interpretation of a program P by a computer [13]. Users can perceive electronic documents, because the computational process generates some physical representations perceivable by them, for example images on a screen, in which texts, pictures and graphs appear. These physical representations only exist and are perceivable until the electronic machinery maintains them in existence. E-documents are less persistent than paper-based ones, but this dependence on a computer offers some advantages. Interactive computers allow e-documents to be managed and adapted by their users more easily than paper-based ones; e-documents also can evolve during their usage and adapt to their users. With the advent of the web, the e-document evolves to "a unit consisting of dynamic, flexible, non linear content, represented as a set of linked information items, stored in one or more physical media or networked sites; created and used by one or more individuals in the facilitation of some process or project" [15]. However, e-documents appear to users as single entities even when their content is distributed in different, geographically remote repositories. Moreover, the physical representation results from a mapping of the content of the document into output events perceivable by users (e.g. the images on the screen or a speech through a microphone). This paper introduces the definition of a user interface description language, IM^{2}L (Interaction Multimodal Markup Language), and of its interpreter. IM^{2}L is an XML compliant language, which permits the specification of e-documents separating the definition and realization of their physical representation from the definition and management of their content.

The paper first discusses the motivations and goals of the project (Section 2), presents a technique to model and specify the e-document (Section 3), and the important annotation operator, which is basic for e-document management (Section 4). These concepts are made concrete discussing in Section 5 a scenario inspired by a case under development. The next Section (Section 6) describes the interpreter implementation and behavior. Last, conclusions are derived (Section 7).

2. MOTIVATIONS AND GOALS

In recent years, the Pictorial Computing Laboratory (PCL) researchers developed a model of human-computer interaction which allows the specification of the process of interaction between human and e-document separating the materialization from the computing process [2]. This separation allows the definition and implementation of adequate control of the human-e-document dialogue based on the ability to capture, trap and manage each action performed by the user on the e-document itself [3]. The separation between materialization and computing implies the separation of the process of creating the content and logical structure of a document from the process of specification of its materialization (physical representation), which may be multimodal. The materialization can be adapted to the culture, skills and abilities of the current user, without altering its content and logical structure.

In this way the traditional notations used in a specific domain to create documents, prescribe procedures and communicate data and results can be assumed as the kernel of the interaction language. The look and feel of the interface therefore reflects the mental models of the users about the task to be performed.

This model stemmed from experiences on the description of the activity of human-e-document interaction in the scientific and technical fields, even if it can be generalized to other interaction activities [12][5]. In these fields, annotation emerged as a first class operator in workgroups performing data analysis and experiments through the web. An experimenter sends annotated results, tools and procedures to his/her coworkers. Annotation is the tool used to make clear the gained insight, the doubts, the questions arising in the work. Coworkers then examine the data, and experiment the tools, mark them up electronically with annotations and return them [5][9].

This model of human-e-document interaction maps naturally into the XML suite approach for organizing interoperable, distributed documents.

On one hand, the XML technology permits the management of distributed contents; on the other hand, the materialization based on SVG (the XML specification for vector graphics) permits the capture and check of each action performed on each pixel composing the physical representation of the e-document. In WIMP systems, an e-document is presented to the user (materialized, in the following) as an image on the screen, whose pixels can therefore be addressed singularly. This approach has two consequences. First also texts are treated as images in the interaction process. Images, graphs, texts, and the mixes of them are treated in a uniform way. Second, the possibility of capturing and checking each action performed on each pixel permits the implementation of: 1) a mechanism for adequate control of the interaction and 2) a mechanism for linking annotation to any element of the physical representation of the document.

An XML-based language, IM^{2}L, has been specified, which admits annotation and interaction control as first class operators. An IM^{2}L program P describes an interactive environment and, can be interpreted by every SVG compliant browser which generates the e-document and controls the interaction process. The browser acts as a table driven interpreter: in this way it can adapt the interaction style to the local user culture.

This approach is similar to the one proposed in [10], in which a rendering engine interprets a UIML (User Interface Markup Language) document [1]. The two approaches differ in the definition of the e-document to be rendered and in the implementation of the rendering machine. Behind the possibility of managing and annotating images, graphic and text in an uniform way and of adequate control of the interaction the IM^{2}L approach has two other features. An IM^{2}L document is a description of an interface which can be plugged in any XML compliant browser equipped with materialization applications. When interpreted by the browser, the IM^{2}L document is transformed into an intermediate data structure (a DOM tree), which is successively materialized by a suitable application. The application is chosen depending on the current context and user. The IM^{2}L document can be materialized according to the user need in a specific context. The hosting system becomes equipped with an interface, which is locally customized to the user culture and skill and can be tailored by the users themselves to their specific needs [7]. Moreover, the content of the e-documents can be materialized as a visual, aural or tactile signal, or a combination of them according to the application adopted for materialization.

3. MODELING AND SPECIFYING THE E-DOCUMENT

In the PCL approach, HCI is modeled as a process in which the user and the computer communicate by materializing and interpreting a sequence of messages at successive instants of time. If we restrict to the case of WIMP interaction [13], the messages exchanged are the whole images which appear on the screen display of a computer and are formed by text, icons, graphs, pictures, windows. Two interpretations of each element on the screen and of each action arise during the interaction: one performed by the user, depending on his/her role in the task, as well as on his/her culture, experience, and skills, and the second internal to the system, associating the image with a computational meaning, as determined by the programs implemented in the system [5]. The user identifies some subsets of pixels on the screen as functional or perceptual units, called *characteristic structures* (css).

Each cs on the screen exists, is made perceivable and evolves because a program exists which is being interpreted by the computer. Each screen layout, and the sequences of screen layout which arise during the interaction come to existence and are made perceivable to humans because of the interpretation of a program. They all are manifestation of virtual entities. A virtual entity (ve in the following) is a system which exists as the result of the interpretation of a program by a computer. A ve is a dynamic system in that it is able to capture the user inputs, compute the reaction to them and materialize its own state - the results of the computation – in a form perceivable by the user.

At each instant, the ve state is defined as a characteristic pattern cp=<cs, u, <intcs, matcs>>, where intcs (interpretation) is a function, mapping ve current cs to the state u of the program generating it and matcs (materialization) a function mapping u to cs.

Users manifest their intentions performing some operation on an input device, which the ve perceives as a set of input events op and refers to a cs on the screen. The pair a=<op,cs>, is called a user activity. The dynamic behavior of the ve appears to the users as a sequence of css generated in reaction to users activities.

A ve becomes an e-document, when some humans use it as a tool of study, consultation or research in achieving a task [11]. An important type of e-document are interactive environments, e-documents whose css are whole images on the screen. An image i on the screen is the materialization of the state d of a program P and is constituted by a finite set of css, each cs_i being associated with the state u_i of a sub-program of P. Two functions, int

and mat relate the css in i to the elements u_i in the description d of the state of P. The triple <i,d,<int, mat>> describes the current state of an interactive environment and is called visual sentence (vs). The set of admissible states of an interactive environment is a set of vss, the visual language we called Environment Visual Language (E-VL).

Each visual sentence must be generated so that it belongs to E-VL. To this end, first a finite set (visual alphabet) of the cp types, whose instances may be composed to form a vs is defined. Next we define on the visual alphabet a Visual Conditional Attributed Rewriting system (vCARW) [3]. A vCARW system contains a set of visual rewriting rules R, which are used to transform a vs_1 into another vs_2, by introducing new cps or modifying existing ones. The rules of a vCARW state how cps of given types can be combined with cps of other types in order to create complex cps up to vss. A Visual Language is specified by coupling a vCARW system with an axiom (vs_0) which is a visual sentence from which the rewriting starts. VCARWs are characterized by rules in which a pictorial and a textual part are made explicit. The pictorial part states the physical appearance – i.e. topology, geometry and shape – of each cs involved in the rewriting step. The textual part makes explicit the computational meaning and operation to be performed. Using the formal specification of a vCARW as a tool for implementation, it is convenient to divide the set R into two subsets: the set of *composition rules* grouping the textual and topological part of the rule, which determine the vs structural organization and the set of *materialization rules*, which determine the shape and geometry of the resulting vs. A finite specification of the interactive environment dynamics is obtained from the following observations: 1) the interaction with an interactive environment always starts from an initial state, a visual sentence vs_0, which is instantiated when the user first accesses the interactive environment; 2) in each state of the interaction a finite number of user activities can be performed; 3) as a consequence of the user activity a, a visual sentence vs_1 is transformed into a visual sentence vs_2 [5]. The designer describes the transformation as $tr:<a,<vs_1 \Rightarrow vs_2>>$, where cs in a=<op, cs> also belongs to the cs of vs_1. The interaction process is specified as a sequence of such transformations. In a transformation, vs_1 and vs_2 share a common part, while the variable part of vs_1 is transformed into the variable part of vs_2 through the application of a *transformation rule* in the form $tr = <a_i, r>$, where a_i is the user activity and r is a rewriting rule of a vCARW. In [5], it is shown that the set TR of transformation rules is finite.

4. ELECTRONIC ANNOTATION AND THE TOOLS SUPPORTING IT

In our current implementation, an e-document is constituted by a body and the tools to operate on it. The body in turn includes data and metadata. Data may be a text, a graph, an image or a mixed of them. Metadata describe properties of the data, such as author, date, procedures followed to create the data. An e-annotation is multimedia-multimodal comment which is added to a part of an e-document, the *target e-document*. E-annotations can only exist associated to a target e-document.

Users create e-annotations with reference to a cs of a target e-document. A user identifies the annotation *base*, i.e. a subset of pixels (some words in a text, a structure in an image) s/he wants to comment within the cs of the target e-document. The user can make evident the base by creating a *visual identifier* and/or make explicit the existence of the annotation by a *visual link*. In any case, the user creates an e-annotation, which can be later retrieved, consulted and updated by the same or other users. Creation, retrieving, updating, and consultation can be performed following different strategies and using different tools. In the scenario described in the next section a stand alone strategy implemented using an *annotation bench* is illustrated [8]. In this strategy, the annotation is materialized by selecting the visual link. As a result, a ve, the *annotation bench*, is made active. The machine needs to know the link between e-annotation and base so that it can retrieve the e-annotation, display it whenever required, use it as an index of the target e-document and update it. On the other side, users need to know if the e-document has been annotated and to be able to manage the e-annotation when necessary.

These activities are allowed by the set of ves described in the following, which a) make manifest to the human the e-annotation, the base and their relations to the target document, and b) permit the construction, retrieving and manipulation of the e-annotation. These ves are here described stressing the interaction point of view. The set of ves consists of:

1. the *visual identifier* is a ve whose css identify the base of the annotation. It can be created by the machine as a reaction to a user action, or by the user. In this last case the target document must be equipped with the tools which permit its creation. The visual identifier is optional. When it is created it may be stationary – when selected it does not react -, reactive, or an active or pro-active widget. In this paper only stationary visual identifiers are used and described, for active and proactive example see [5][6].
2. the *visual link* is a ve that links the base to its annotation. Its cs is an icon, materialized near or pointing to the base. The selection of its current cs determines the activation of the annotation bench which permits the management of annotation data. The visual link is created by the e-document, as a reaction to the user action of selection of an annotation button. The user has also to indicate the point where it must be placed. Its cs is defined at the document design time - for a rhetoric study of its design see [4].
3. the *annotation bench* is a complex ve, which allows the creation and updating of a set of annotations that users consider logically related. Its cs displays the annotation, (possibly) some metadata on the annotation and the set of tools through which the user can create the annotation and interact with it. The annotation and metadata can be

represented as textual notes, images, pictures, graphs, or a mixture of them [5][6].

5 A SCENARIO

To make concrete the above considerations, we provide here an example describing how a team of glaciologists and photointerpreters reach the classification of a remote sensed image, by the exchange of electronic annotations. Photo-interpreters and glaciologists interpret the image using two prototypal interactive environments, B-Glaciologist and B-Photointerpreter, which allow management, editing, processing and annotation of target e-documents. The specification of the two environments has been performed by following the approach described in Section 3, through which the definition of the corresponding E-VLs have been obtained. Some elements of B-Glaciologist E-VL, and their relations and dynamics, are described in the following. The environments share a knowledge repository, in which e-documents and annotations are stored. The environments run under a web browser (Internet Explorer in this case), and therefore environments and repository may reside in (possibly) different places and can be used at (possibly) different time. Photo-interpreters and glaciologists report the results of their activity and their observations as annotations, which can be independently accessed and improved by all the participants to the activity. In this implementation, annotations are displayed on the screen by an *annotation bench*, an e-document whose body is constituted by a set of annotations and related metadata (in this case the prototype environment in which the annotation is generated, title of the annotation, number of the annotation at hand, total number of the annotations associated to the base in the current thread, title of the target e-document). Fig. 1 displays a screenshot of B-Glaciologist, representing the current state in an interpretation process. In the screenshot, the Explorer tools cs can be recognized at the top of the figure. Under them a header identifies the team of system developers. Three cs lie under the header: an equipment area on the right with a title identifying B-Glaciologist, a working area on the left, and a message area on the bottom. In the equipment area, three menus are present for the management of repositories of entities to be worked (images and annotations) and equipments to work on entities. In the working area, the cs of a target e-document and an annotation bench are present. The user (a glaciologist) is annotating a target e-document, formed by a body and a set of tools to work on it. The body is constituted by the data (a raster image in our case), and metadata, the name of the geographical entity ("Gruppo Adamello"), the name of the current user ("Mario Bianchi") analyzing the image of interest. Tools are represented by icons in a toolbar on the top of the body. Among the tools, the button labeled 'a' is the cs of a visual link creator and the button labeled with the closed curve of a visual identifier creator .

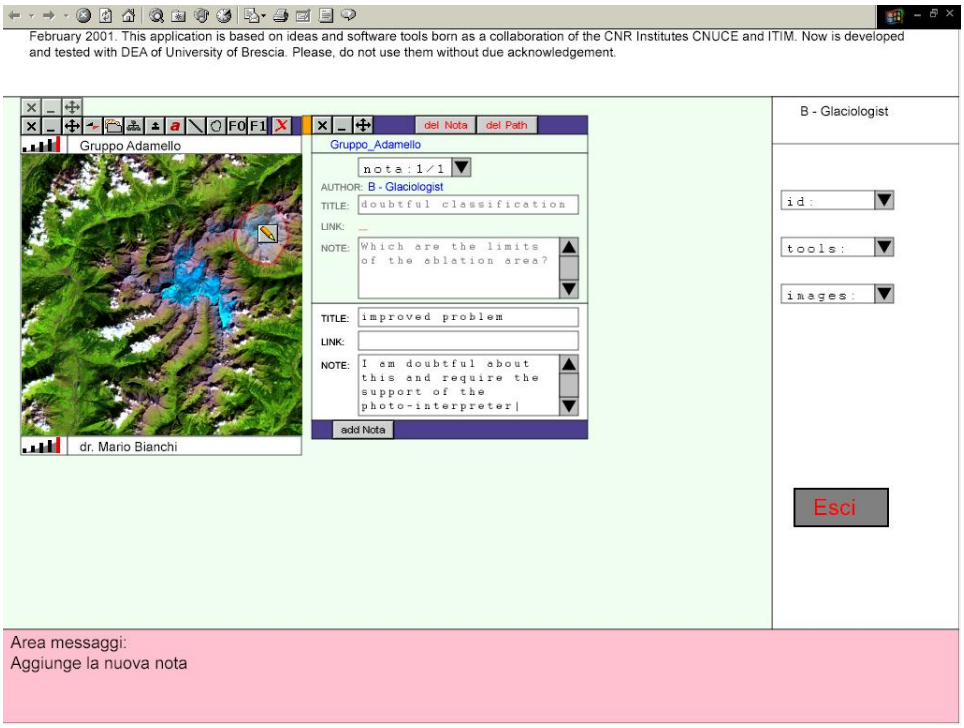


Fig. 1: Interacting with B-Glaciologist the second glaciologist is seeing the annotation performed by the first glaciologist and is adding his own annotation.

The screenshot is the result of the following process. A first glaciologist has recognized a structure of interest in the body data of the target e-document, and marked it as a *visual identifier*, the opaque shield surrounded by a closed red line. To this end he selected the visual identifier creator icon. B-Glaciologist visual identifier creator reacted to this selection displaying a cursor – a cross - which the glaciologist used to trace the boundary of the base. The visual identifier creator generated the opaque shield within the boundary. Next the glaciologist created a first annotation, by selecting the visual link creator and then a pixel within the visual identifier. B-Glaciologist visual link creator reacted to this selection creating a visual link ve, whose icon is the pencil and activating an annotation bench. Visual link and visual identifier are created as two ves, which become components of the target e-document. The visual identifier cs is created at run time by the user as a graphical SVG entity, while the visual link cs is a predefined bitmap. These two css are superimposed to the cs of the target e-document. The annotation bench displayed its cs, a form which the first glaciologist filled composing his annotation. Then he saved the annotation by clicking the "add note" button and left the environment. In the current situation the glaciologist Mario Bianchi retrieved the annotation and target document from the archive and is replying by adding his own annotation. Note that the annotation bench css displays its current state. Under a toolbar and an header identifying the target e-document, two different forms are materialized. The first in turn shows a menu and an annotation body .The menu allows the selection of one of the existing annotations: its caption displays the number of the currently displayed annotations and the number of exiting ones. The annotation metadata are the author and title, the data of the annotation are in this case a text – could be a graph or an image or a mix of them. This first form permits the exploration of set of annotations which resume the history of the current task. The second form collects the data (text in this case) and metadata (title) provided by the user: Metadata about the author are automatically derived by the annotation bench. The two forms constitute the cs of an annotation group. If the user thinks that he needs to report a different thread of reasoning on the annotation base, he can open a second group by restarting the visual link creation procedure. At the end of this annotation activity, the glaciologist may store the annotation in the knowledge repository by selecting the 'add Nota' (add note) button. Later the photo-interpreter can access the annotated target e-document using B-Photointerpreter. The target e-document conveys its visual identifiers and visual links, from which the photo-interpreter can retrieve the e-annotations [8].

6. AN XML IMPLEMENTATION

In this section we outline the architecture underlying the two prototypal interactive environments, with the aim of explaining the choice of the XML technology [16]. Moreover, we show how the use of SVG (the XML specification for vector graphics [17]) permits to see each element of the e-document and e-annotation as a graphical entity, which therefore can be managed as such. This feature is exploited to permit the capture and check of each action on each pixel composing the image on the screen and the tailoring to the user needs.

6.1 The system architecture

Figure 2 sketches the architecture of the system: an SVG compliant browser interprets an IM^{2}L program. IM^{2}L (Interaction Multimodal Markup Language) is an XML-compliant language, whose markup encodes a description of the ves' storage layout and logical structure [14]. IM^{2}L is designed to allow a proper interaction with multimedia and multimodal data. To this end, data are embedded into virtual entities equipped with a set of tools which support the desired interaction. An IM^{2}L program describes a whole interactive environment [6][7]. An IM^{2}L program is composed by: a) one or more IM^{2}L fragments, which define the content and logical structure of the interactive environment; b) one or more fragments expressed by using languages suitable for materialization; c) one or more fragments written in a suitable programming language, which specify the dynamics of the interactive environment and are suitable for web-based interaction. In the current implementation, sketched in Fig. 2, an IM^{2}L program is constituted by:

- A *Starter SVG*. It is the starter system linking the IM^{2}L document with its interpreter.
- An *IM^{2}L document*, composed of one or more IM^{2}L fragments specifying the contents and the logical structure of the ves composing the interactive environment and of the interactive environment itself.
- The *DTD of IM^{2}L*, defining the grammar to compose the elements of an IM^{2}L document. The grammar is composed by a vocabulary containing the atomic elements of the Visual Language, and the composition rules determined in the specific user domain. It thus implements the visual alphabet of cps and the composition rules defined at the specification step.
- The *DTD of SVG*, defining the composition rules of the elements of an SVG document.
- The *set of SVG prototypes*, specifying the physical materialization of the ves and their topological relations.
- The *set of javascript instantiation functions*, which implement the materialization rules defined at the specification step. They instantiate the SVG prototypes with the information included in the IM^{2}L document and compute their materialization features, such as geometry, color, appearance. The result of the instantiation is used by the viewer to establish the image i on the screen.
- The *set of javascript interaction managing functions*, which implement the transformation rules of virtual entities defined at the specification step. They compute the topological and geometrical features of virtual entities whenever their appearance or position must be modified. They also compute the reactions of ves to the user activities. An IM^{2}L program is interpreted by an SVG

compliant browser. The browser coordinates the interpretation of the input events (user activities) according to the IM^{2}L program. Such interpretation is performed by a standard XML processor [16], an interpreter of the used programming language, and a standard SVG viewer (in our case the Adobe SVG Viewer plug-in). The next section describes how the browser instantiates the initial state of the interactive environment, a necessary introduction to the description of how interactive annotation creation and evolution.

6.2 Interactive environment initial state instantiation

The first user access results into the system initial state (vs_0) instantiation. This process is illustrated in fig. 2 as the path of arrows with ordinal numbers.

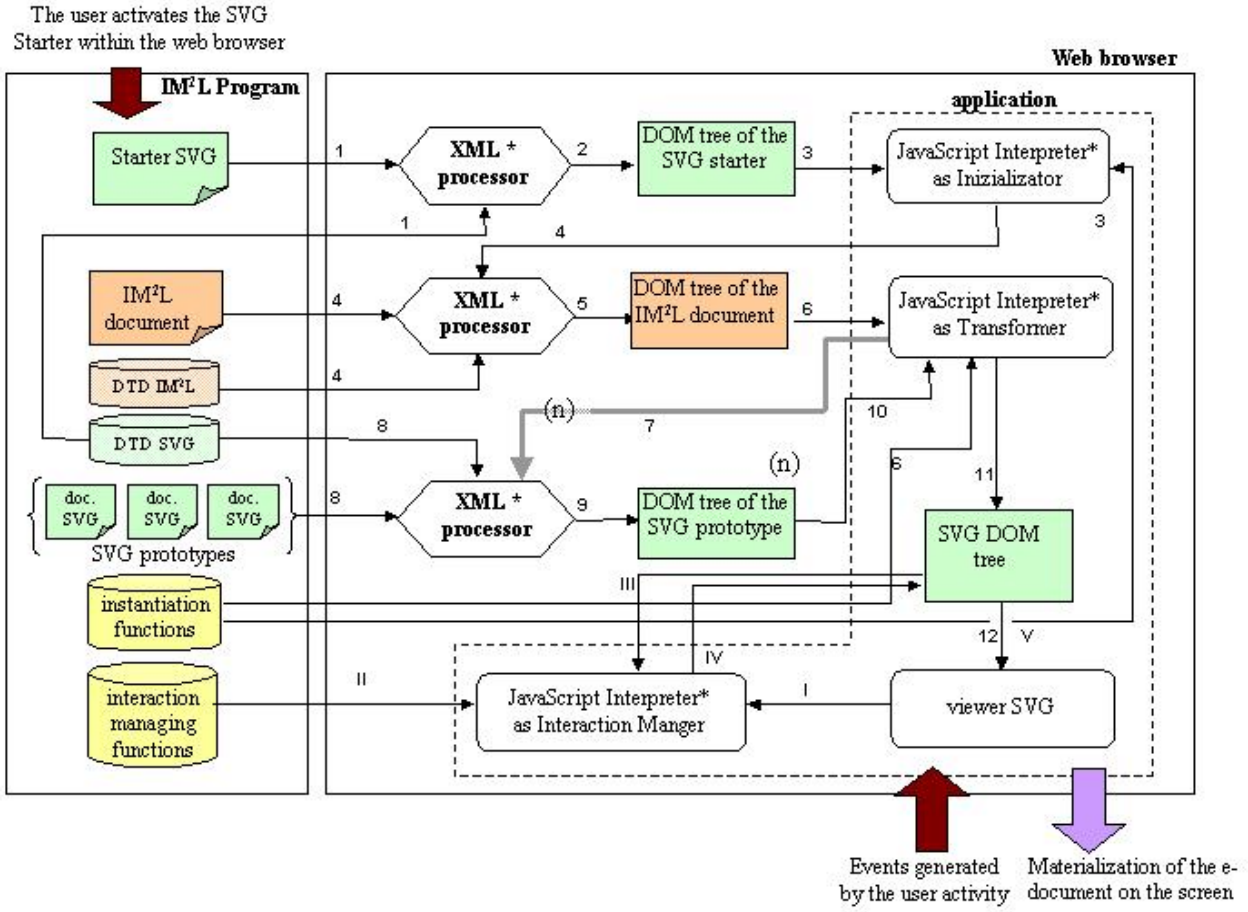


Fig. 2– The interaction between the user and interactive environment: ordinal numbers indicate the steps in the creation of the vs_0; roman numbers the steps in the management of an interaction event which causes the modification of the DOM tree ('(n)' indicates that a step is repeated n times ; '*' indicates that the icon represents a same program used in different steps of the process on different parameters).

The steps in the process are:

1. The user loads the SVG starter within the browser. This document is processed and validated by the XML processor using the DTD of SVG.
2. The XML processor creates the DOM (Document Object Model) tree of the SVG starter.
3. The DOM tree of the SVG starter is analyzed by the inizializator, which loads the IM^{2}L document.
4. The IM^{2}L document is loaded in the web browser and is processed and validated by the XML processor using the DTD of IM^{2}L.
5. The XML processor creates the DOM tree of the IM^{2}L document.
6. The DOM tree of the IM^{2}L document is an input for the transformer (a javascript interpreter).
7. The transformer fires the XML Processor, asking for the set of SVG prototypes corresponding to the virtual entities in the DOM tree of the IM^{2}L document.
8. Each SVG prototype is processed and validated by the XML processor using the DTD of SVG.
9. The XML processor produces a DOM tree fragment for each SVG prototype.
10. Each DOM tree fragment of an SVG prototype is instantiated by the transformer using the information included in the DOM tree of the IM^{2}L document.
11. The transformer embeds each DOM tree fragment of the SVG prototypes into the full SVG DOM tree.

12. The full SVG DOM tree is the input of the SVG viewer that materializes the state of the interactive environment on the screen.

6.3 Interaction with the interactive environment to create an e-annotation

The SVG DOM tree is a data structure that represents the hierarchy of virtual entities whose css are currently visualized. The viewer interprets it to materialize the image i representing the current state of the interactive environment on the screen. Within such tree, some nodes, called here *ve nodes*, correspond to virtual entities, while the child nodes of ve nodes may correspond to attributes of the virtual entities or to virtual entities (and therefore they are in turn ve nodes) composing the considered ones. The user creates an e-annotation with reference to a target e-document. Whenever the user performs an activity to create an e-annotation, the viewer captures the events generated by the activity, and sends them to the javascript interpreter, which calls the interaction managing functions associated to the events. These functions modify the DOM tree: new nodes are added as child nodes of the sub-tree describing the target e-document being annotated, more precisely, the ve node associated with it. This node is the root of the sub-tree which specifies the target e-document and its elements. Figure 3 shows this node after several annotation activities have been performed on only one base. Users reported different threads of reasoning, creating M groups of e-annotations. Each group in turn includes several annotation bodies. In this situation, only one visual identifier exist, while M visual links have been created to permit independent access to each thread of reasoning. The child nodes created by the annotation activities are: a) ve nodes, i.e. the *visual identifier* and the *visual links*; b) the node *annotation data,* which is the root of a sub-tree containing the information on the annotation bodies. The child nodes of the node *annotation data* describe *annotation groups*, each one linked to a visual link (number 2 in figure 3). An *annotation group,* in turn, is the root of a sub-tree, whose child nodes are *annotation body* nodes. An *annotation body* node has two child nodes: *data* and *metadata*. *Data* may include images, text and graphic entities properly combined. *Metadata* contain data about the annotation, such as the author, the title, and a possible link to an url or a file. In the case of visual identifiers and visual links creation, new virtual entities are generated at run-time. This corresponds to the creation of new ve nodes in the DOM tree. For each new virtual entity, an IM^{2}L document fragment must be loaded and processed to be included as a new node within the full SVG DOM tree materialized by the viewer. Due to space limitations, we do not describe here this process. Details about the interactive creation of virtual entities and the integration of the corresponding nodes into the full SVG DOM tree can be found in [14]. On the other hand, annotation data have a different nature: they are associated to the target e-document being annotated, but are accessed, interpreted and made manifest by an annotation bench, whose corresponding node is already present in the DOM tree while the user is interacting with the system to insert (or delete) his/her annotation. Therefore, these data are stored as a sub-tree rooted in *annotation data* node. The process of creation and updating of *annotation data* is illustrated in fig. 2 as the path of arrows with Roman numbers:

I. The event generated by the user activity (e.g. typing a character to compose the content of the annotation) is captured by the SVG viewer and passed to the interaction manager.
II. The interaction manager calls for the execution of the interaction managing function related to the user activity.
III. The interaction manager interprets the function.
IV. The interaction manager updates the overall SVG DOM tree, by: a) if this is the first character, creating the sub-tree necessary for storing annotation; and b) modifying the attribute *Data* of the interested annotation body.
V. The modified SVG DOM tree is the new input of the viewer**,** which is materialized on the screen.

The last process is the creation of the link between the visual identifier and a visual link. When the user selects a point to be the annotation anchor within the visual identifier, the system creates a visual link, and pro-actively creates a link between the visual identifier and the visual link. Therefore, a link between the corresponding nodes is created in the DOM tree (number 1 in figure 3). A visual link may exist without being related to any visual identifier, for example, when the user has selected a point in the target e-document not included in a visual identifier.

7. CONCLUSIONS

IM^{2}L is a user interface description language in that an IM^{2}L program defines an interactive environment and its interface. An XML processor, in our case embedded in a browser, can interpret an IM^{2}L program. The result of the interpretation is a new document in an intermediate language, for example the SVG DOM tree in Fig. 2. This new document can be interpreted by every application capable of managing and materializing documents in the intermediate language; in our case, this role is played by the SVG viewer. On the other side, the IM^{2}L document describes the interactive environment – and hence its interface -independently from the style of materialization, the device of materialization and the materialization mode (visual, aural, haptic). The materialization style and modes are embedded in different IM^{2}L program fragments, in the example the SVG prototypes and javascript instantiation functions. The application, enriched by the plug-in specified in the program (in the example the DTD of SVG) translates an IM^{2}L document into a document in the language for which a materialization application exists. Exploiting the modularity of the W3C approach, and using other open-source tools available on the net, a first prototype of the kit that manages e-documents and e-annotations has been developed as an interoperable and off-the-shelf system.

ACKNOWLEDGMENTS

The authors wish to thank: A. Rampini of IREA-CNR, who provided the case study; M. F. Costabile and her group for the stimulating discussions during the development of this work; R. Gentile and E. Villani who developed the prototypes.

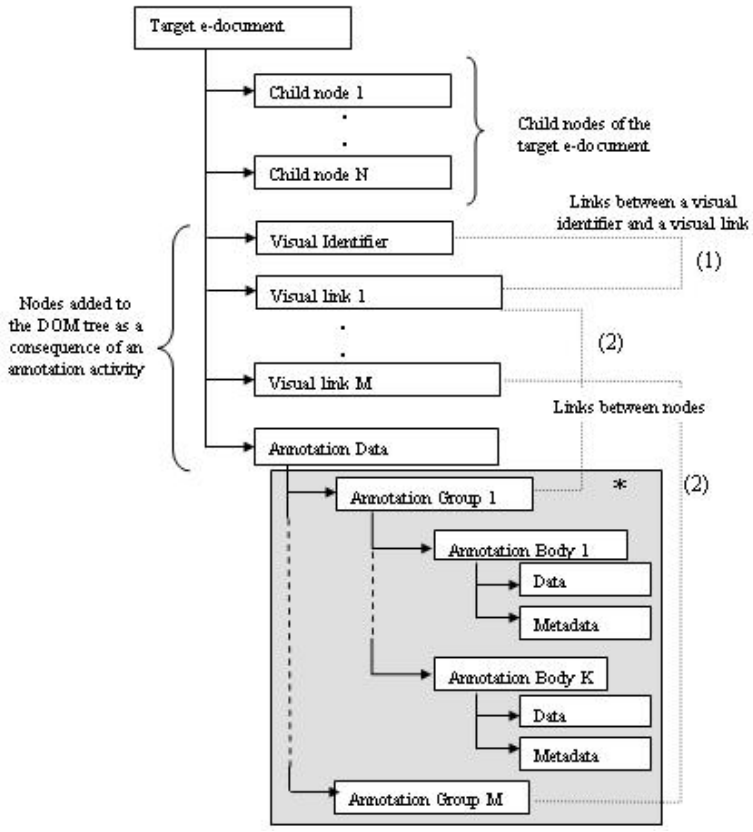


Fig. 3– DOM tree fragment at a certain instant t: it represents the state of the annotated target e-document, to be interpreted by the viewer. Ordinal numbers indicate the links described in the text

REFERENCES

1. Abrams, M., Phanouriou, C., Batongbacal, A., L., Williama, S., Shuster, J., E., UIML: An applicance-Independent XML User Interface Language, *WWW8 / Computer Networks*, 31(11-16), 1999, 1695-1708.
2. Bottoni, P., Costabile, M. F., Levialdi, S., Mussio, P., Defining Visual Languages for Interactive Computing, *IEEE Transactions on SMC*, 27(6), 1997, 773-783.
3. Bottoni, P., Costabile, M. F., Mussio, P., Specification and Dialog Control of Visual Interaction. *ACM TOPLAS* 21(6), 1999, 1077-1136.
4. Bottoni, P., Levialdi, S., Rizzo, P., An Analysis and Case Study of Digital Annotation, *Proc. 3rd Int. Workshop DNIS 2003*, Aizu, Japan, LNCS 2822, 2003, pp. 216-231.
5. Carrara, P., Fogli, D., Fresta, G., Mussio, P., Toward overcoming culture, skill and situation hurdles in human-computer interaction, *Int. J. Universal Access in the Information Society*, 1(4), 2002, pp. 288-304.
6. Costabile, M.F., Fogli, D., Fresta, G., Mussio, P., Piccinno, A., Computer Environments for Improving End-User Accessibility", *Proc. of 7th ERCIM Workshop "User Interfaces For All"*, Paris, October 23-25, 2002, pp. 187-198.
7. Costabile, M. F., Fogli, D., Fresta, G., Mussio, P., Piccinno, A. Building Environments for End-User Development and Tailoring, *Proc. 2003 IEEE Symposia on Human Centric (HCC'03),* Aukland, New Zeland, October 2003, 31-38.
8. Fogli, D., Fresta, G., Mussio, P., On Electronic Annotation and Its Implementation, *Accepted at AVI 2004*, May 2004.
9. Heck, R., M., Luebke, S. M., Obermark, C. H., A survey of Web Annotation Systems. *Digital Documents* 1999, Id. 31.
10. Luyten, K., Coninx, K., UIML.NET: An Open UIML Rendere for the .Net Framework, *Proc. CADUI 2004*, Funchal, Madeira Island (Portugal), 2004.
11. Mussio, P. E-Documents as tools for the humanized management of community knowledge, *Keynote Address, ISD 2003*, Melbourne, Aug. 2003.
12. Mussio, P., Finadri, M., Gentini, P., Colombo, F., A bootstrap approach to visual user-interface design and development, *Visual Computer*, 8, 1992, 75-93.
13. Preece, J., Rogers, Y., Sharp, H., Benyon, D., Holland, S., Carey, T. *Human-Computer Interaction*, Addison-Wesley, Wokingham, UK, 1994.
14. Salvi,D., Progettazione di ambienti integrati per la produzione di ambienti interattivi, Thesis, Univ. di Brescia, Italy, 2003.
15. Shamber, L. What is a document? Rethinking the concept in uneasy times. *Journal of the ASIS*, 47 (9), 1996, 669-671.
16. W3C Consortium, Extensible markup language (XML), 2001, http://www.w3.org/XML.
17. W3C: Scalable Vector Graphics (SVG), [Online] 2001 http: //www.w3.org/Graphics/SVG

Extending XML UIDLs for Multi-Device Scenarios

Elmar Braun[*], **Max Mühlhäuser**
Telecooperation Group
Department of Computer Science
Darmstadt University of Technology
Alexanderstr. 6, 64283 Darmstadt, Germany
{**elmar, max**}**@tk.informatik.tu-darmstadt.de**

ABSTRACT
Device independent user interface description languages are used to create concrete user interfaces for a multitude of devices from a single abstract user interface description. All current languages have in common that the target device for the concrete user interface is assumed to be a single self-contained device. But what about ubiquitous computing environments, which are generously equipped with large number of publicly available devices that are associated on demand by users who roam among devices? In this case the set of devices currently near the user forms a virtual target device. This virtual device can have unusual properties. For example, it can have more than one screen. It also changes when the user moves in or out of context of a device. We explore how current XML-based single authoring languages can be extended to support such scenarios.

INTRODUCTION
Many user interface description languages (UIDL) have been motivated by the need for device independence. The advent of a multitude of mobile devices (PDAs, cell phones, ...) has made it necessary to provide several different user interfaces for a single application: one for each device type on which users might conceivably want to use the application. "Handcrafting" the user interface once per device incurs a prohibitive amount of effort for design and development. The reuse of user interfaces that were written for a specific device is not possible. Since the targeted devices can vary greatly regarding their means of interaction (e.g. graphical vs. voice-based, large vs. small screen, normal vs. numerical vs. no keyboard, ...), a user interface which is well designed for one device can be unusable on another.

What is required is *single authoring*: one *device independent* description of a user interface, which can automatically be adapted to the specifics of all conceivable target devices. This goal has sparked a considerable number of research efforts. While some have researched device independent widget toolkits (e.g. [9, 12]), many have focused on XML-based markup languages (e.g. UIML [1]).

If an authoring scheme is truly device independent, it should be possible to generate a user interface for *any* conceivable target device with it. That includes device types which the developers of the authoring scheme did not consider in their design, either because the developers were unaware of them, or even because they had not been invented yet. In the research domain, *ubiquitous computing* [21] confronts us with such a scenario, in which unusual devices meet a strong need for single authoring. Our ubiquitous computing research explores the concept of *federated devices*: rather than accessing applications through a single full-featured device, they are rendered on a set of several associated devices. For example, a user could use a private minimal voice-only headset device in conjunction with a publicly available display, which was associated on demand. We will explain such ubiquitous environments, and their challenges regarding authoring of user interfaces, in more detail in the next section.

The need for single authoring is quite obvious in this scenario. The set of target devices, for which a user interface needs to be provided, has grown from all full-featured standalone devices to all feasible combinations of devices. This diversity makes single authoring and automated generation of concrete user interfaces inevitable. But traditional single authoring schemes were not developed with such device groups as the "virtual target device" in mind. Our work in progress investigates how current single authoring schemes can be applied to such scenarios, and which extensions they need in order to transcode for device federations. Instead of reinventing the wheel by developing a new single authoring language, we have decided to extend an existing markup language (subsets of XHTML [4] and XForms [8]). The use of XML allows easy extension of the existing markup with tags and attributes from another namespace. We are also developing software which orchestrates multiple devices of various modalities, so that we can test the productions of our single authoring and transcoding components.

UBIQUITOUS COMPUTING
Before judging, and possibly extending, existing single authoring approaches for ubiquitous computing settings, the

[*]The author's work was supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the Graduiertenkolleg (Research Training Group) "Systemintegration für ubiquitäres Rechnen in der Informationstechnik".

traits and requirements of such settings need to be explored more closely. Mark Weiser described ubiquitous computing as "enhancing computer use by making many computers available throughout the physical environment, but making them effectively invisible to the user" [21]. Our research focuses on two aspects derived from this quote. The first is exploring ubiquitous computing environments which are amply augmented with interactive devices. The second aspect is to allow sessions to move from device to device, following the user, in order to avoid impairing user mobility and convenience by binding them to a fixed device.

Public and Shared Devices

The first part of the quote advocates a more relaxed relationship between users and devices. Instead of today's model of exclusively personal devices, computers are envisioned to be pervasively woven into the infrastructure, and anyone who has need for their services can associate them. In the ubiquitous environment which our group is building, that includes interactive devices like displays. A practical example of this is a display in a public space: any user who has need for visual access to some information can walk up to such a display, be automatically associated with it (because the infrastructure detects the user's presence), and start accessing personalized information and applications.

Of course, ubiquitous computing does not dispute the need for personal mobile devices. On the contrary, it can "radicalize" the concept of mobile devices. Current mobile devices need to have satisfactory input and output capabilities; they cannot be minimized much further because their usability would suffer too much. But if the user is able to associate interactive devices from the environment on demand, the isolated use of a mobile device only occurs in the rare case that the user is nowhere near an ubiquitously augmented space. Consequently, its interactive capabilities can be stripped down to a bare minimum of "last resort" abilities. Some researchers have gone as far as to propose mobile devices completely without any means of interaction, which solely rely on associating devices from the infrastructure [20].

Our group is developing a voice-only headset called *Talking Assistant* (TA) [3]. When worn, simple commands or information queries can be quickly activated by voice; it is neither necessary to disengage hands and eyes from their current occupation to operate a PDA-like device, nor to walk up to and associate the nearest public screen. But for more complex interactions, the user is expected to use a public screen or pick up some other nearby device. When the user does that, the TA does not go dormant. It forms a federation with the other device, which can now be controlled by voice through the TA, and thereby achieves a multimodal presentation.

The second part of Mark Weiser's quote ("making them effectively invisible") challenges us to make the use of associated devices as natural as possible. Associating a device should neither force the user to perform a lengthy login or reconfiguration process, nor should it require restarting running applications. Ideally it should not require any conscious user action at all. We are experimenting with various user tracking technologies to detect user intention. The TA includes head position and bearing trackers, which can be used to detect which device a user is looking at. Association with a screen can be performed by detecting a prolonged glance at that screen.

Session Mobility

Effortless dissociation of devices is as important as effortless association. A user should not be bound to a device once it is associated. The user might want to move to a different device that is better suited to her needs, or leave a fixed device because she needs to go to some other place. Requiring the user to explicitly save and close a running application, and restore its state on the next device she uses, would not be "invisible". Instead it should be possible to roam between devices at runtime, "carrying" running applications from device to device. Applications are made mobile by making sessions mobile, rather than making the device on which they run mobile.

This is also referred to as *teleporting* [7]. The original teleporting works by forwarding a user's desktop to whichever computer she happens to be near. Obviously this only works within one modality and device type: from desktop computer to desktop computer. Combining this with user interfaces authored in a device independent manner allows teleporting a running application between different devices and modalities, e.g. from a screen to a voice-based device.

SINGLE AUTHORING FOR MULTI-DEVICE INTERFACES

A single-authoring language for multi-device UIs shares several basic requirements with the single device case. One example is a device independent representation of widgets (e.g. "select one" instead of "radio button"). We decided to base our experiments on the existing work of one of the many available XML single authoring languages. The advantage of an XML language is the simple extensibility. We decided to use use subsets of XForms [8] and XHTML [4]. While other single authoring languages may be better in certain aspects, we wanted to base our extensions on rather "basic" language. We wanted to avoid that these extensions become dependent on some special features of the underlying UIDL, so that they are general enough to be used in other UIDLs as well. We may evaluate how easily we can apply our concepts to a different UIDL in the future, to see whether this goal has been met. In addition, HTML has the advantage that it is well-known, there are many available tools for processing it, and transcoding to real HTML is simple, which allows us to use standard web browsers as clients (see below).

Our extensions take the form of a number of additional tags and attributes[1]. We will explain the most important elements below.

[1]Note: the XML examples are simplified for the sake of readability. This includes lacking namespaces, and using simplified and shortened tag an attribute names, which differ from those used in our language.

Placing Widgets

The two main problems with multiple channels and devices is choosing on which channel to render a widget, and deciding whether to render it on multiple channels concurrently. For example, consider the use of a handheld PDA, which offers easy to reach input capabilities (because it usually is in the hand of the user), in conjunction with a large wall mounted screen. This combination shall display a user interface consisting of a large text and a number of buttons for navigating to other pages of the text. How should the elements of the user interface be distributed to the devices? The text should be rendered on the wall mounted display only, as it would be difficult to read on the PDA, while the buttons go on the PDA, where they can be pressed. Even if the wall mounted display has touch interaction, the user would have walk up to it to press a button, while the PDA can act as a remote control.

How is this expressed in markup[2]? We assign an importance attribute, or weight, ranging from 0 to 1, to each widget. When rendering, we use this attribute to place more important widgets first on devices that are easier to interact with.

```
<button value="1.0"/>
<p value="0.5">Long text...</p>
```

This however does not take into account that these widgets have a different value regarding input and output. Therefore these can be specified separately. The other form still is available as a shorthand for widgets with equal weight for input and output.

```
<button valueIn="1.0" valueOut="0.0"/>
<p valueIn="0.0" valueOut="1.0">...</p>
```

The weight is also used to determine which items to drop if the space on a channel is limited. For example, if there were a large number of buttons, not all would fit on the PDA. The highest-valued would then be displayed on the PDA, while lower-valued buttons would overflow to the wall mounted screen if it has touch interaction capabilities. (Otherwise the only possibility is putting all buttons on the PDA and force scrolling.) This effectively creates a remote control with the most used buttons, while allowing access to the full functionality on the wall screen.

Duplicating Widgets

In a single-channel UI, each element of the UI has to be rendered exactly once. Rendering more than once on the same channel has little benefit, while rendering a widget not at all would make the UI inoperable. In an environment where the user interface is rendered on multiple channels concurrently, it may make sense to render the same element more than once. One example is multimodality: the same widget is rendered in different modalities concurrently. In the input case this has the advantage that the user can pick to do input through the modality she feels most comfortable with. In the output case it has the advantage that the combined presentation through multiple channels may be easier to grasp than a single-modal presentation. On multiple channels of identical modality, it is usually not reasonable to show a widget multiple times, since the user normally can focus at at most one device at a time. However, if the button from the above example has a visible state, it might make sense to display it on the large screen too, so that the user does not have to look at the input device only to verify its state.

The weight from the last paragraph is also useful as a hint whether an element is sufficiently important to justify duplicating it or not. However, there is also an explicit syntax to influence this behavior:

```
<button replicate="..."/>
```

Setting this attribute to `once` prevents duplication, whereas `always` tries to render a widget multiple times even on identical modalities.

Grouping Widgets

Obviously automatically placing widgets to different devices incurs the danger of splitting up groups of related widgets, which should be displayed close to each other. Therefore one of the most important tags is the `<group>` tag, which allows to bind related tags together. For convenience it is also possible to attach attributes to the group tags, which are then inherited by the enclosed widgets.

Privacy

Since we assume frequent use of public devices in ubiquitous computing environments, privacy is an issue. Private data should not be displayed on a large public display, even if it was the best available device. Widgets can be marked as private (never shown publicly), shared (made public if possible, useful for limited cooperative work support), or non-sensitive (no preference). This attribute can be set at runtime; such elements default to private, but can be toggled to shared. The transcoder creates an additional private button, which when activated switches the view to reveal the previously private data. Furthermore we have two levels of "public": one that uses any public device, and one that uses public devices only within a trusted environment like a private office.

Interim Conclusion

This list of additional tags and attributes may appear to be rather short. This is not only because of its work in progress state. While some less important tags have not been listed here (e.g. requesting a specific modality for a elements of the user interface that make only sense in that modality), this list covered the most important ones. Since our goal was to single-author multi-device interfaces with as little additional effort as possible, having few additional tags and attributes, which cause additional authoring effort, is quite positive, as long as they are sufficient to create multi-device interfaces.

[2] Actually for this example, no additional markup is needed. The transcoder detects that the text requires much screen space and is output only, while the buttons need to be rendered on a device where they can be pressed, and require no visual feedback if they have no state. However, for the sake of an easy example, it is presented with explicit markup here.

Whether these hints work well depends on how the transcoder evaluates them. So far we have found them to be sufficient to create reasonable multi-device interfaces. However, in future versions it shall be possible to pass new transcoding rules to the transcoder in the application markup, in order to achieve better than default user interfaces at the expense of additional authoring effort. This is discussed in the next chapter.

RENDERING ON MULTIPLE DEVICES AT RUNTIME

In order to test that user interfaces specified with our UIDL transcode into reasonable usable interfaces, we have to render and test them in an actual ubiquitous environment. We are currently building a room equipped with several large and small screens, as well as other input and output devices, embedded in the infrastructure. Besides the necessary hardware infrastructure, we also have developed software which orchestrates all those devices. Rendering a user interface with this runtime environment comprises of a number of steps.

The first step is discovering which devices are in the user's vicinity. For this purpose we use a user tracking system [2], and correlate the user location to a world model (a database of device locations). Another method we use are badges worn by each user, which detect transmissions from nearby infrared-emitting tags that are fixed to each device. While less exact, this method has the advantage of not needing a world model.

Finally, while the application is running, we need to synchronize the input and output among the federated devices.

Distributing to Devices

In the second step the runtime must decide which devices should render which parts of the UI, and transcode the device independent UI representation into a format that the devices can process. The rules for picking devices and distributing elements of the user interface often look like this: "if there is a speech device and a screen device, then ..."; or " if there is a large screen and a small screen, then ...". The conditions in these rules refer to the available devices and their properties (e.g. "is a speech device" or "screen x is larger than screen y"). In order to express and evaluate these conditions, the devices must provide a description of their capabilities. Currently we use simple (self-written, not vendor-provided) descriptions based on the Composite Capabilities/Preferences Profile (CC/PP) [14], but we may replace it with a language of our own in the future.

The rules itself are currently hard-coded in Java in the transcoding engine. This is an unsatisfactory way of expressing them, since it is hard to add new rules, or expand and tune existing rules. This is why we plan to express such rules as markup as well. This would make extensions easier, and allow for user-defined preferences (e.g. a blind person could specify never to use screens), as well as application specific rules.

One might ask whether such rules should be necessary at all. Should not the UIDL alone deliver sufficient information to pick devices and perform the transcoding? In our experience, the problem with specifying rules that refer to particular devices is that the resulting markup is hardly device independent, that they require much effort to specify common sense rules again and again for each user interface, and that it is easy to forget to provide rules for a particular device or device combination. Therefore we deliberately designed our UIDL to consists mostly of hints to the transcoder, rather than tags and attributes with a rigid deterministic behavior. Instead, optionally additional transcoder rules can be embedded in the UIDL. This gives the user interface author some control over the authoring effort / quality tradeoff. The author can trust the default rules of the transcoder to generate reasonable results from the UIDL alone, while for better quality or for frequently used device combinations a custom rule can be added.

Transcoding for Specific Devices

The output of the transcoder, that distributes elements of the user interface to the devices, is not yet device specific. Instead it generates for each device a document in an intermediate format, which again consists of subsets of XHTML and XForms. This document consists of those parts of the source document that have been selected to be rendered on this particular device. In a second transcoding step, a device specific user interface is generated from the intermediate document. Our group has developed a sophisticated method for such transcoding from a device independent representation to a concrete user interface (see [12]). However, currently we simply use web browsers for rendering our user interfaces (see below). This means that the last transcoding step is transcoding from pseudo-HTML to HTML, which does not require such a sophisticated transcoding scheme. It mostly consists replacing XForms with with HTML forms (due to lack of browser support), and adding styling.

Synchronizing Devices

Now that we have distributed the different elements of the UI to the devices, they can render their respective portion of the UI. For that purpose the devices need client software which performs the rendering. For our first prototype, we have decided to use a web browser as the renderer, because most of our target devices have a browser available.

For our desktop and large screen client we implemented a wrapper around the Internet Explorer. As explained above, transcoding into the native format simply means transcoding to HTML with form elements as input widgets. The browser performs rendering and interaction, and the wrapper is responsible for starting the browser when a user steps up to a screen. The wrapper also closes the browser when the user leaves the screen, and communicates with the other federated devices (see below). A speech client, which wraps IBM ViaVoice in a similar fashion, is currently under development.

Once the UI is rendered on multiple federated devices, these devices need to be coordinated. When a user makes an input, she should get some feedback that the input has been

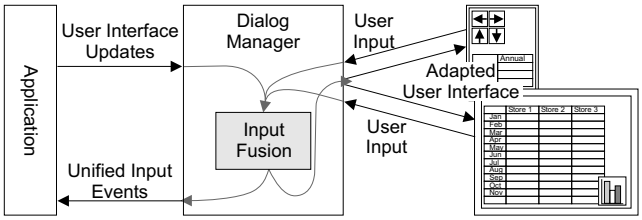


Figure 1: The dialog manager synchronizes input and output events among several devices and the application.

recognized correctly. Input and output of the feedback need not occur through the same channel. For example, when a user makes speech input through her headset, an associated display should update the respective input field with the recognized value.

For the coordination, all clients immediately send all input they receive to a server which we call *dialog manager* (DM, see Figure 1). The DM forwards this input to all other clients, which update their view accordingly. In other words, the DM holds the model in a MVC pattern with multiple distributed views and controllers. The DM is also where the actual application connects to the UI. In our case, the application is a Java class that receives events about input from the DM, and can make changes to the model, causing the DM to send out the appropriate updates to all clients.

EXAMPLE APPLICATION

To help understanding our work, we will demonstrate a simple audio player UI as an example. An audio player is a good example of the type of user interface we are investigating. While using the application and listening to music, the user will only sporadically focus on the UI of the player. Therefore the user cannot be expected to permanently be in front of a full featured device like a desktop PC. Yet it should be possible to quickly acquire control of the player. And if the user so desires, the player should be able to provide detailed additional information about the music being played.

```
<p valueIn="0" valueOut="1">
  <em>Now Playing:</em>
  Night on Bald Mountain
<p>
<p valueIn="0" valueOut="0.5">
  <em>Artist Info:</em>
  Modest Mussorgsky, born into a ...
<p>
<select1 valueIn="0.5" valueOut="0.5">
  <item>Night on Bald Mountain</item>
  <item>Pictures at an Exhibition</item>
  <item>Boris Godunov</item>
  <item>...</item>
</select1>
<group valueIn="1" valueOut="0">
  <button>Play</button>
  <button>Pause</button>
  <button>Stop</button>
  <button>Next Song</button>
  <button>Previous Song</button>
  <button>Increase Volume</button>
  <button>Decrease Volume</button>
</group>
```

This UI consists of four main parts: The first simply displays the title of the currently playing song. The second part displays a longer text with information about the artist. These two elements have no value as input elements. Regarding output, the information about the artist is considered less important than the currently playing title. The third part is a playlist, which lists all songs that are enqueued to be played, and allows the user to select a title to jump directly to it. This element is both valuable for input (“select a song”) and output (“show which songs are next”). The last part is a group of buttons, which allow to control the most frequently used functions of an audio player: starting and stoping, skipping forward or backward, and volume control. These widgets have no informative value, but are input controls of high importance.

How would this interface be rendered on a combination of a PDA and a larger display? The PDA displays primarily the buttons, and acts as a form of remote control. Besides the buttons, its screen also has space for rendering the most important output element and therefore displays the song title (see Figure 2(a)). The large display has space to show the entire interface (see Figure 2(b)). Whether it displays the buttons depends on whether it also has input capabilities (e.g. a touch screen or through an attached mouse). Figure 2(b) assumes an output-only display.

RELATED WORK

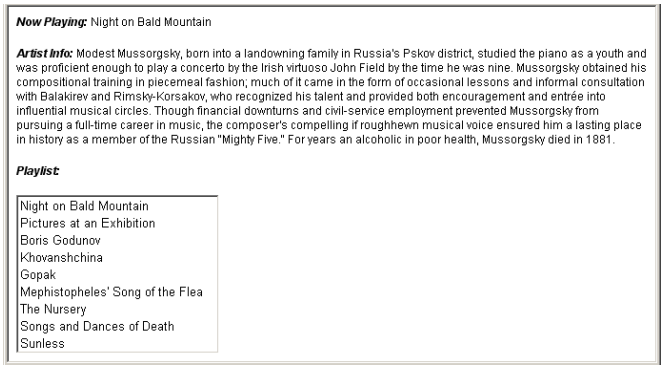
Since single authoring for multi-device interfaces has not received much scrutiny yet, the related work can be categorized in two groups: work that deals mostly with multi-device interfaces but not with single authoring, and work that deals with single authoring but only for single-device interfaces.

Multi-Device Interfaces and Browsers

The use of portable and stationary devices together has been explored before (e.g. [19, 15]). However, such research usually focused on exploring one particular application on one particular fixed combination of devices. Our work provides single-authoring for an arbitrary set of devices, determined at run-time.

Since our prototype is mostly based on web technology, it resembles a multimodal browser that uses different devices for the different modalities. Browsers which span more than one device have been explored before in several projects. The development of a multimodal browser by synchronizing browsers on multiple devices has been described in [13]. Pages for this browser are authored by manually creating multiple versions of the same page in HTML, WML, and VoiceXML. Single authoring was not considered. The similar WebSplitter [11] also synchronizes multiple browsers, but does not support non-visual modalities. It features a single-authoring approach, albeit with a different goal: dif-

(a) The PDA part of the audio player UI.



(b) The large screen part of the audio player UI.

ferent views for different user roles (e.g. teacher, student) are created from a single source, and then co-browsed synchronously by multiple collaborating users.

Other multi-device browsers do not focus on multimodality or teleporting between devices, but on remote controlling other associated devices from one central device. The UbicompBrowser [6] runs on a handheld device, and controls nearby devices through specialized URLs. For example, the URL `tv://local/station` causes a nearby TV to show the channel "station". Interaction with this system always takes place through the handheld; associating nearby devices to render a better user interface was not among the goals of the UbicompBrowser. The similar Small Screen / Composite Device (SS/CD) project [18] renders multimedia on multiple federated devices, which are automatically selected from the devices available in the user's environment. Rendering multimedia output is considerably different from rendering a user interface, and has no need for single authoring: the source consists of media in a particular modality (e.g. video), and transcoding this to another medium (e.g. audio) is in most cases neither sensible nor possible.

Bandelloni and Paternò [5] propose a system which uses single-authoring and supports runtime migration of web applications between different device types. To our knowledge the concurrent use of multiple devices is not their goal, unlike the other projects mentioned here.

Single Authoring

The problem of single authoring in a device independent manner has received much attention in the past, both in the form of device independent widget toolkits [9] and number of markup languages like UIML [1]. To our knowledge, none of these deal with multiple federated devices. However, Göbel et. al. propose, along with their own UIDL, an algorithm which partitions a user interface into several parts, for the purpose of displaying them on multiple successive pages [10]. This algorithm might be adapted to partitioning a user interface into parts that are rendered concurrently rather than successively on several devices.

Other projects like XWeb [16] approach single authoring by expressing an application's model, rather than view and controller, in a canonical device-independent way. Each client device should have its own XWeb client (comparable to a web browser), which knows which widget best represents a particular piece of the model. Olsen et. al. did consider rendering a XWeb interface on multiple devices concurrently, but did not provide the devices with a specialized UI for this case [17]. To our understanding, each device simply renders its stand-alone UI even when used concurrently with others.

XHTML [4] and XForms [8], which we have used, are limited as device independent UIDL, because, besides other issues, the only type of user interface they can describe are forms. So far this limitation has been acceptable for us, since we explore primarily simple user interfaces that are used casually. Complex user interfaces (e.g. with multiple concurrent windows) usually require the full attention of the user due to their complexity, and are unlikely to be used in such a casual manner while the user is roaming about from device to device.

CONCLUSIONS

We have presented our work in progress on building user interfaces that are rendered across multiple devices and modalities concurrently. Using our methods, applications can escape the limitations of a single device, while not requiring considerably more authoring effort than with current single-device single-authoring languages.

Besides a single authoring scheme for such multi-device interfaces, we have shown how to discover and assemble such device federations, how to generate useable user interfaces for them, and how to synchronize multiple devices at runtime with our dialog manager. Future work will include user studies to fine tune and verify our transcoding scheme.

REFERENCES

1. ABRAMS, M., PHANOURIOU, C., BATONGBACAL, A. L., WILLIAMS, S. M., AND SHUSTER, J. E. UIML: an appliance-independent XML user interface language. *Computer Networks (Amsterdam, Netherlands: 1999) 31*, 11–16 (May 1999), 1695–1708.

2. AITENBICHLER, E., AND MÜHLHÄUSER, M. An IR Local Positioning System for Smart Items and Devices.

3rd International Workshop on Smart Appliances and Wearable Computing (2003), 334–339.

3. AITENBICHLER, E., AND MÜHLHÄUSER, M. The Talking Assistant Headset: A Novel Terminal for Ubiquitous Computing. Tech. Rep. TK-02/02, Fachbereich Informatik, TU Darmstadt, 2002.

4. AXELSSON, J., EPPERSON, B., ISHIKAWA, M., MCCARRON, S., NAVARRO, A., AND PEMBERTON, S. XHTML 2.0. `http://www.w3.org/TR/2003/WD-xhtml2-20030506`, May 2003.

5. BANDELLONI, R., AND PATERNÒ, F. Platform Awareness in Dynamic Web User Interfaces Migration. In *Proceedings of MobileHCI 2003* (2003).

6. BEIGL, M., SCHMIDT, A., LAUFF, M., AND GELLERSEN, H.-W. The ubicompbrowser. In *Proceedings of the 4th ERCIM Workshop on 'User Interfaces for All'* (1998), ERCIM.

7. BENNETT, F., RICHARDSON, T., AND HARTER, A. Teleporting - Making Applications Mobile. In *Proceedings of 1994 Workshop on Mobile Computing Systems and Applications* (Santa Cruz, USA, 1994).

8. DUBINKO, M., KLOTZS, L. L., AND RAMAN, T. V. XForms 1.0. `http://www.w3.org/TR/2003/REC-xforms-20031014/`, Oct. 2003.

9. GELLERSEN, H.-W. Modality Abstraction: Capturing Logical Interaction Design as Abstraction from "User Interfaces for All". In *Proceedings of the 1st ERCIM Workshop on 'User Interfaces for All'* (1995), ERCIM.

10. GOEBEL, S., BUCHHOLZ, S., ZIEGERT, T., AND SCHILL, A. Device Independent Representation of Web-based Dialogs and Contents. In *Proc. of the IEEE Youth Forum in Computer Science and Engineering* (Nov. 2001).

11. HAN, R., PERRET, V., AND NAGHSHINEH, M. WebSplitter: A Unified XML Framework for Multi-Device Collaborative Web Browsing. In *Proceedings of ACM CSCW'00 Conference on Computer-Supported Cooperative Work* (2000), pp. 221–230.

12. HARTL, A. A WidgetBased Approach for Creating Voice Applications. In *Proceedings of MobileHCI* (Udine, Italy, 2003), pp. 7–10.

13. KLEINDIENST, J., SEREDI, L., KAPANEN, P., AND BERGMAN, J. Loosely-coupled approach towards multi-modal browsing. *Universal Access in the Information Society 2*, 2 (June 2003), 173–188.

14. KLYNE, G., REYNOLDS, F., WOODROW, C., OHTO, H., HJELM, J., BUTLER, M. H., AND TRAN, L. W3C recommendation: Composite capability/preference profiles (CC/PP): Structure and vocabularies 1.0. `http://www.w3.org/TR/2004/REC-CCPP-struct-vocab-20040115/`, Jan. 2004.

15. MYERS, B. A. Using Handhelds and PCs Together. *Commun. ACM 44*, 11 (2001), 34–41.

16. OLSEN, JR., D. R., JEFFERIES, S., NIELSEN, T., MOYES, W., AND FREDRICKSON, P. Cross-Modal Interaction using XWeb. In *Proceedings of the ACM Symposium on User Interface Software and Technology* (2000), Toolkit Support for UI, pp. 191–200.

17. OLSEN, JR., D. R., NIELSEN, T., AND PARSLOW, D. Join and Capture: A Model for Nomadic Interaction. In *Proceedings of the 14th annual ACM symposium on User interface software and technology* (Nov. 2001), ACM Press, pp. 131–140.

18. PHAM, T., SCHNEIDER, G., AND GOOSE, S. A Situated Computing Framework for Mobile and Ubiquitous Multimedia Access Using Small Screen and Composite Devices. In *Proceedings of the 8th ACM international conference on Multimedia* (Marina del Rey, USA, 2000), ACM Press, pp. 323–331.

19. REKIMOTO, J. A Multiple Device Approach for Supporting Whiteboard-Based Interactions. In *Proceedings of ACM CHI 98 Conference on Human Factors in Computing Systems* (1998), pp. 344–351.

20. WANT, R., PERING, T., DANNEELS, G., KUMAR, M., SUNDAR, M., AND LIGHT, J. The Personal Server: Changing the Way We Think about Ubiquitous Computing. *Lecture Notes in Computer Science 2498* (2002), 194–209.

21. WEISER, M. Some Computer Science Issues in Ubiquitous Computing. *Communications of the ACM 36*, 7 (July 1993), 74–84.

Adaptation for Device Independent Authoring

Guido Menkhaus and Sebastian Fischmeister
Software Research Lab, University of Salzburg, Salzburg, Austria
{Menkhaus, Fischmeister}@SoftwareResearch.Net

ABSTRACT
The impact of device independent authoring on software engineering manifests itself mainly at the middleware level. Until recently middleware platforms were targeted at vertical coverage of specific scenarios. Consumer devices with integrated Internet-access are becoming more popular and their diversity grows with their market penetration and with the extension of the mobile communication infrastructure. This requires software architectures that are capable of supporting horizontal coverage of a wide range of devices and scenarios. This paper presents the Multi User Interface, Single Application project. It provides a feasable approach for multi-platform support through the introduction of an adaptable and abstract interaction-oriented user interface language.

Author Keywords
Device Independent Authoring, User Interface Adaptation

INTRODUCTION
Due to the emergence and proliferation of new classes of devices accessing services on the Web, device independent authoring became an important issue. The vision of device independence authoring is to allow services and content on the Web to be accessible by *anyone, anywhere, anytime, and anyhow* [21] by simultaneous reuse of authored source across multiple contexts and environments. Because of the variety of different UI platforms, content authors cannot afford creating and developing content and UIs that target only one out of a set of target platforms. This has a profound impact on the way UIs are build. Systems must scale up well to environments that include a wide variety of different devices that can easily and flexibly connect to application logic. The objective is to develop UIs for the same application only once and not for each particular class of computing device to avoid fragmentation of the web space into spaces that are solely accessible with specific type of devices.

There are research projects that look into new generation UIs that no longer consist of a display, but for example of wearable glasses projecting the UI onto the eye of the user [9] or wearable computers with small and semitransparent displays placed only centimeters from the user's eye in front of the line of sight [24]. These UIs, although small in physical size, will no longer have size constraints concerning the UI. However, user acceptance seems to be low [2]. We think that traditional UIs still have a strong potential for improvement and that this technology will prevail in the near future on the consumer market [11].

The dynamics of the mobile environment and the limitations of mobile computing resources make adaptation a necessary technique. Adaptation is necessary when there is a significant mismatch between the supply and demand of a resource, which is typical for a mobile and pervasive computing environments [18]. A permanent solution therefore requires models and techniques that allow UI designer to generate adaptive UIs.

This article presents the Multi User Interface, Single Application (MUSA) system. The MUSA project concentrates on multi-platform support. We argue that the introduction of an abstract interaction-oriented UI language is an essential component that eases the development of UIs for mobile computing devices. MUSA allows a Web-based service to be delivered to a variety of target platforms without additional effort. When a user requests a service, context information triggers the adaptation mechanism tailoring the event handler graph (written in EGXML) and the content of the service to a target platform specific presentation form. The event handler (EH) graph mediates between the concrete UI and the application logic. Figure 1 sketches the scenario of device independent authoring with the EH graph.

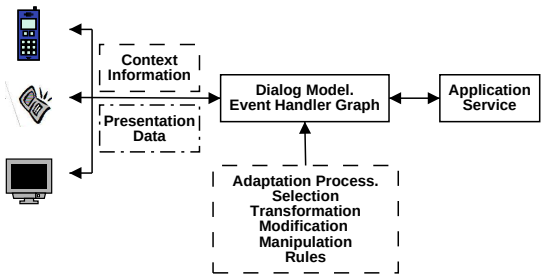


Figure 1: Scenario of device independent service authoring.

The remaining of the article is organized as follows: The following section presents a short overview of UI architecture

and related work. We then introduce the MUSA system and model. The adaptation mechanisms of the EGXML is discussed and results are presented. We close the article with concluding remarks and a short discussion about our future work.

USER INTERFACE ARCHITECTURE AND RELATED WORK

Model-based UI software development has introduced concepts and techniques that assist in the process of UI development and a large number of layered architectures have been devised [15, 20]. Myers for example, has identified four general layers: window, widget, view, and model [13]. This division corresponds roughly to the linguistic model of architecture of interactive software that distinguishes between the following three layers: The semantic layer describes the tasks users should be able to perform using the application for which the UI provides the interaction means. This layer corresponds to the model in Myer's layered architecture. The syntactic layer describes the structure and the interaction behavior of the UI. This can be mapped onto Myer's view component. The lexical layer consists of the detailed description of the visual part of the UI and corresponds to the window and the widget layer of Myer's architecture.

The Arch-Slinky model refines the level of abstraction with which it describes the reference model for UI management systems [23]. It proposes a structural and functional decomposition into the following five components:

1. **Functional Core.** The functional core is the creator of data that the system represents. It manipulates data and performs other domain-oriented functions.
2. **Functional Core Adapter.** The functional core uses the functional core adapter as its channel of communication to the dialog component.
3. **Dialog.** The dialog component mediates between domain specific and presentation specific data. It controls task sequencing.
4. **Logical Interaction.** The physical interaction and the dialog component interact through the logical interactor that provides corresponding interfaces and objects.
5. **Physical Interaction.** The physical interaction component consumes the presentation specific data and provides input to the functional core. It deals with input and output of data on the target device.

The Arch-Slinky model minimizes the effect of future modification to an application and its environment by isolating the dialog component from the functional core and the physical interaction component.

There are a number of approaches for designing and implementing UI software. They range from the automatic generation of the presentation model from a more or less formal task model [3] to informal, structured guidelines on how to build UI software [10]. However, lots of effort has been dedicated to approaches that can be placed somewhere in the middle of the both extremes [1, 6, 17]. Common to these approaches is the application of a single XML-based description implementing the presentation and/or the dialog model, respectively. The description is adapted according to a device profile at run-time into a device adequate form. The approach supports device independent authoring so that a single description is enabled to serve a multitude of platforms. Wong presents a high-level task model description of a web application that has a tree-like structure [22]. The tree-like structure is adapted according to device-dependent information to match the target device. Adaptation on the task and concept and the UI level is presented in [8]. Rules are defined and prioritized that tailor a UI at different levels for graceful degradation. Most of the work concentrates on transformation of UI elements. Adaptation at the task and concept level focuses on deletion and insertion of tasks.

The introduction of a custom platform-independent markup language can help to solve the problem of the *Tower of Babel* in UI languages, since one obstacle to device independent authoring is that each platform with its typical browser has its own markup language and each language aims at a specific platform and is optimized for supporting it. However, the support of different platforms is a problem that can be solved at the physical interaction / UI level of abstraction. Another main obstacle to device independent authoring is the growing number of networking enabled devices with a wide variety of UI capability and device specific platforms. One of the main differences they share is different screen size. How to enable content to be adapted to various screen sizes? The same content may require varying numbers of windows to display and a different navigational structure, depending on the platform. For example, content fitting on one PC window may require three windows on a mobile phone. Yet, all these windows originate from the same, single authored UI. The device profile delivers information about the limitations and restrictions of the target platform. The adaptation process for EGXML exploits this information for adapting the content and the navigational structure. This results in a hierarchical structure of dialogs.

MUSA

The MUSA system utilizes device independent UI description in EGXML and supports the integration and composition of Web services. The objective is a reduction of development time, cost, as well as improved maintainability and flexibility.

Figure 2 illustrates the high-level architecture of MUSA. The system is conceptually split into four tiers and employs an event-driven design.

Client: The client environment represents the first tier. The client is represented by a device with a UI. No service data is installed on the client side and the client communicates via wireline or wireless Internet with the service. Typically, the client is some sort of browser, but in principle, could also be a device with no visualization capacity such as a telephone.

Request Processor and Client Gateways: The communication between the service and the client passes through the

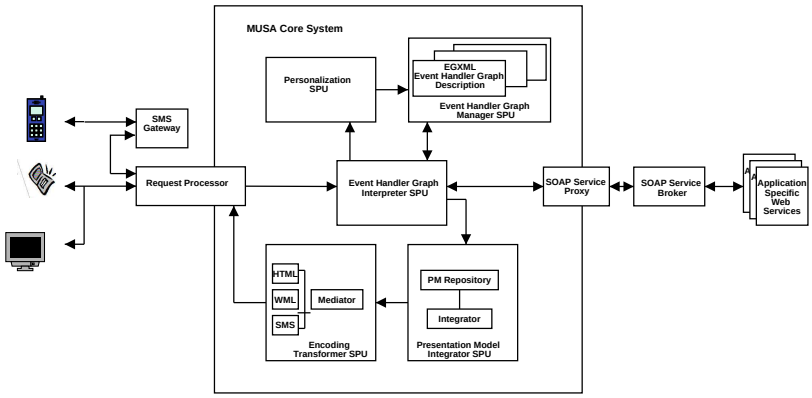


Figure 2. High-level architecture of MUSA system.

request processor. It forwards the communication stream to the MUSA core system and converts the client requests into events that are used throughout the MUSA architecture.

MUSA Core System: The MUSA core system consists of specialized processing units (SPU), which reflect the separation of concerns of the system.

1. **Event Handler Graph Interpreter SPU.** The EH graph interpreter handles the event processing. The incoming events from the request processor are sent to individual EHs that contain the necessary information to properly respond to the input event. In response to the event processing the system generates outgoing events for each incoming event, which are further processed in the MUSA system.

2. **Event Handler Graph Manager SPU.** The EH graph manager SPU manages the EH graphs. It controls the access and the transformations of the EH graph.

3. **Personalization SPU.** The personalization SPU enables users to personalize, i.e., to modify and adapt the EH graph to their preferences. The adaptation is done via direct manipulation technique [19]. For example, EHs can be removed from the EH graph. Once an EH is removed, the removal can be undone. EHs can be given a different description and they can be preset with default values. This is helpful, when a user applies a form over and over again, and the text of only a few input fields varies. It helps reducing the time to fill a form.

4. **Encoding Transformer SPU.** The transformer SPU transforms and maps outgoing EHs of the EH graph to an appropriate presentation form. If the client is a device with a graphical UI, the EHs are mapped to those concrete UI elements, which are able to implement and trigger the specified events, which are associated to specific EHs. The SPU applies a transformation on the EH graph depending on the client's profile. Figure 2 shows three transformers: a HTML, a WML, and a SMS transformer.

5. **Presentation Model Integrator SPU.** The integrator SPU models the overall presentation layout of the EH graph, which are transmitted in the course of the current interaction between users and applications. The presentation model integrator SPU has a repository of presentation models. The presentation models are created at design time by a UI designer. This allows the EH graph and the presentation models to be developed, maintained, and modified independently. A presentation model in the repository consists of a file written in a concrete UI language enriched with special *integrator commands*, which indicate where to integrate the EHs of the EH graph. An opening command opens an integrator command and each opening command has a corresponding closing command, which delimits it, such as a XML element has an opening tag and a closing tag. Each concrete UI element, which is between the opening and the corresponding closing command, belongs to this integrator command. The insert command indicates the place where to insert the EH of the EH graph. If the associated EH is not present, the complete command and its content is removed and not transferred to the encoding transformer SPU. The simplest presentation model consists solely of integrator commands.

Service Proxy: The service logic is the body of code for which the MUSA system provides the service facade. The Web services that implement the service logic are accessible via service proxies, which connect the MUSA system to other Web services.

MUSA Model

MUSA builds on the Arch-Slinky model adopting the cardinal functional decomposition in a physical interaction, logical interaction, dialog, functional core adapter and the functional core component. The MUSA model (Figure 3) refines the Dialog component, by introducing the MUSA EH graph and the Physical interaction component, by splitting it into a global and local component. The MUSA Model tries to map the abstraction a Web designer would use while designing a Web-based service onto the vocabulary of the EH graph. Within the dialog component, we observe two kinds of information flows: the vertical traversal of the EH graph and the lateral information flow with the functional core adapter and the logical presentation component. An EH of the EH graph may be related to one or multiple objects of the func-

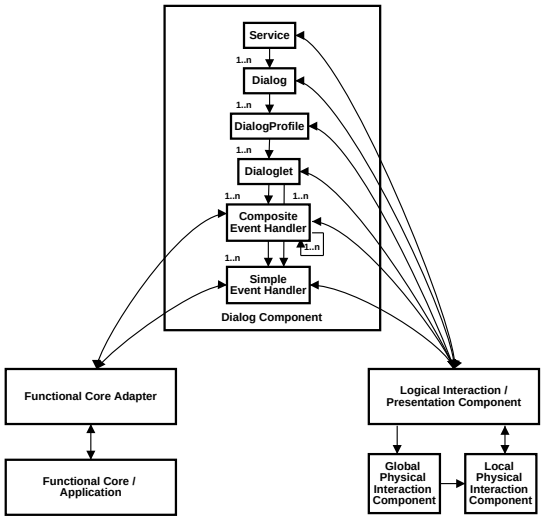


Figure 3. MUSA Model.

tional core adapter. Similarly, each part of the EH graph is connected to one or multiple presentation objects of the logical interaction component.

The physical interaction component is split into a global and a local part. This allows the both parts to vary independently. This avoids a permanent binding of the encoding of the objects of the logical interactor and the global layout of the logical interactor to each other.

The MUSA Event Handler Graph (EGXML)

The EH graph is at the core of the MUSA model. The introduction of the EH graph follows the idea of the reactive constraint graph described in [4]. It is an abstract description of a service logic, which is available for service access to a wide range of clients. The basic building blocks of the EH graph are represented by specific EHs. The EHs receive events from the client dispatched by the logical interaction component and emit events in response to the event processing. In case of a client with a UI, outgoing events are assigned and eventually mapped in the physical interaction component to concrete UI elements that are able to trigger the corresponding EHs. The UI elements trigger the event either on display of the UI elements or in response to user interaction.

The objective of the concept of the EH graph is to structure the service design by using the abstractions Service, Dialog, Dialoglet, Composite and Simple Event Handler. Each of these plays an important role in service UI design in practice. By providing the EH graph for describing these abstractions, the vocabulary of the designers informal design practices is matched. This makes it easy for the designer to map its vocabulary to the abstractions, both in terms of formalizing an informal specification and communicating the results to other stakeholders.

The EH graph as an implementation of the dialog component runs inside an EH graph interpreter and contains the description of the service logic in EH graph XML (EGXML). It handles the event sequencing and processing. The following hierarchical structures help the designer to organize the service logic into a dialog model.

1. **Simple Event Handler.** An EH is an abstract interaction object. It contains the necessary information on how to handle an event coming from a UI and to delegate the processing of the event and its associated data. It is a concrete UI object's target and represents it from a behavioral point of view.
2. **Composite Event Handler.** An EH is composite, if it is composed of other EHs.
3. **Dialoglet.** A dialoglet consists of a number of EH, which belong to one group – logically and semantically.
4. **Dialog Profile.** A Dialog Profile consists of a device profile and one or more dialoglets.
5. **Dialog.** A dialog is designed to represent a task or a sub task of a specific Web-based service. A dialog contains one or more dialog profiles. A dialog profile represents the dialog through the filter of a specific device profile. A dialog is composed of an initial dialog, from which other dialogs are chained.
6. **Service.** A service is composed of a sequence of dialogs.

During the design of the EH graph, the difficulty consisted of the support of a wide variety of possible UIs for the access of web-based interactive services. The least sophisticated format determines the features of the EH graph. This effect is also known as the least denominator problem. The EH graph format incorporates elements that can be transformed to equivalent elements in all target formats. For example, the graphical UI of a service intended for a desktop computer may be quite different to a UI that is appropriate for a mobile telephone with a very small display. However, although the concrete UI elements are quite different for each target device and the layout mechanisms vary, the interaction mechanisms are similar.

EGXML ADAPTATION

The definition of a single dialog model is still oriented at the "one device - one functionality" paradigm, but today we can access mutually any service through any device [5]. This requires an appropriate mechanism to dynamically adapt the dialog model. In this article a dialoglet of the dialog model will be represented by a two-dimensional discrete function $e(x, y)$, which is digitized both in spatial coordinates and feature value: $dialoglet = [e(x, y)]_{P \times Q}$ where $P \times Q$ is the size of the dialog, (x, y) denotes the spatial coordinate and $e(x, y) \in EH$ the type of EH of the EH graph from the set of available EHs EH. Without loss of generality we consider only the case, where $Q = 1$.

Clustering EHs that implement the dialog model into a hierarchical structure of dialoglets is the essential step in our adaptation process that leads to device independent authoring (Figure 4). For this, the dialog model adaptation process partitions the EHs implementing the dialog model into

non-intersecting dialoglets such that each dialoglet satisfies a homogeneity predicate. We consider the case, where the current dialog model was intended to be displayed on a device like a desktop PC with a monitor and the actual device that accesses the service is a PDA or mobile telephone with a much smaller screen. This situation is typical for mobile computing: Services target primarily desktop PC with a monitor and latter are ported to a wide variety of mobile computing devices. The situation in which a service targets small devices and is accessed by a desktop PC with a monitor is not further discussed here.

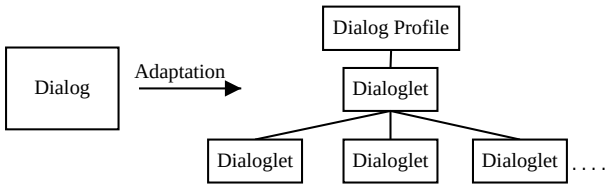


Figure 4. Dialog adaptation.

Formally, the process of adaptation of the dialog model can be defined as follows: If a dialog model consists of a set of EHs and P is a homogeneity predicate, then the adaptation of the dialog model is a partitioning of EHs into a set of connected dialoglets $(d_1, d_2, \ldots, d_n)$, which will eventually be converted into a hierarchical navigationable structure of dialoglets, such that:

$$\begin{aligned} dialoglet &= \cup_{i=1}^{n}(d_i \backslash e_{navigation}(d_i)) \\ d_i \cap d_j &= \emptyset, i \neq j \\ d_i &\text{ is a connected set of event handlers} \\ P(d_i) &= \text{true}, i = 1, \ldots, n \\ P(d_i \cup d_j) &= \text{false, if } d_i \text{ is adjacent to } d_j \end{aligned}$$

A user accessing a service supported by a dialog model needs to navigate from one dialoglet to the next dialoglet. However, not all navigation elements are in the original dialog. Thus, they have to be integrated into the dialoglets resulting from the adaptation process. The set of all UI elements in the dialoglets equals the EHs in the original dialog plus the integrated new EHs dedicated to the navigation between the dialoglets, the $e_{navigation}(d_i)$.

EGXML Adaptation using Event Handler Clustering

The above definition of the process of adaptation is very similar to image segmentation as defined in [16]. Analogous to segmentation and clustering processes, the more context and domain information is known beforehand and integrated into the process, the better the process' results.

Approaches exploring dialog model adaptation can broadly be divided into two categories. Processes of the first category do not consider context knowledge such as screen size during design time. They work bottom-up and rely uniquely on dynamic adaptation of the dialog model. The other category explicitly uses top-level domain and task model knowledge during design time. The processes are configured with a priori known target contexts. The quality of the latter approach depends on the configuration and the type of content that is presented. The former approach has the drawback of working only on syntactic information. We propose a hybrid approach that combines the advantages of both approaches, fast design, no need to produce sophistic configuration data and integration of semantic information.

The two main challenges of the hybrid approach to dialog model adaptation are: How to incorporate low-level semantic information into the dialog model? How to adapt the dialog model respecting the semantic information? Our adaptation technique is based on a linking strategy of two hierarchies of graphs [12, 14]. It allows remodeling a dialog of the dialog model into dialoglets of connected EH and the use of low-level task model information.

The elements of the dialog model are placed as EHs into a stack of regular grids, as illustrated in Figure 5. In the lowest level of the stack, each cell of the grid corresponds to a single EH. Each cell of level $i + 1$ represents a group of cells of level i. The adaptation algorithm always forms linear structures of 3×1 cells. The cells overlap in such a way that the outer cells on level i belong to two cells of level $i + 1$. The cells in a group of level i, represented by a cell of level $i + 1$, are called the subcells or the children of this cell. The representing cell is called the parent of its children.

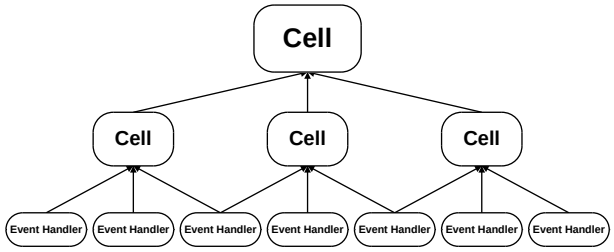


Figure 5: Stack of a regular grid of cells that places a structure on a set of EHs. Three EHs form a cell on the lowest level. Cells on a lower level are candidates for cells on a higher level.

The clustering of a set of EHs into a set of dialoglets is done within the boundaries of the induced stack of cells and is of primary interest. To come to the final set of dialoglets, we dynamically build up a stack of EH-regions. An EH corresponds to a EH-region on the lowest level. Adaptation of a dialog is performed by clustering EH-regions of level i into EH-regions of level $i + 1$. However, EH-regions can only be grouped within the boundaries of a cell in which they reside, as illustrated in Figure 6, and if they satisfy the homogeneity predicate. This guarantees that we cluster only connected and adjacent EH-regions. Complete EH-regions, i.e. regions that cannot further be clustered, result in dialoglets.

The framework to describe the adaptation technique is the description as a hierarchy of graphs. The first hierarchy of graphs forms a syntactic based and static structure that guarantees that the resulting dialoglets are connected. The second hierarchy is dynamically built up respecting the low-level semantic information integrated into the dialog model

at design time. The two hierarchies of graphs implement the dialog model adaptation process. The process consists of the following four phases:

- **Bottom-up Clustering.** EH-regions of level i are grouped into EH-regions of level $i + 1$ within the boundaries of their cell and satisfying a predicate P.
- **Top-down Separation.** EH-regions that fail to group on level i are separated recursively down to level 0.
- **Horizontal Separation.** Large-sized EH-regions of level i, especially when they contain a single EH, are split.
- **Relinking.** The user should be able to navigate from one dialoglet to the next dialoglet. To ensure usability, EH-regions are relinked by integrating additional navigation EHs.

Bottom-up Clustering: The clustering process determines the set of connected EH-regions of level i of a specific cell and groups them. In order to form a new region r_{i+1} (the subscript indicates the level) in a cell c_{i+1}, the set of subcells are determined. Each subcell has a set of regions associated that are candidates for grouping into r_{i+1}. A region s_i groups into the region r_{i+1}, if it satisfies the homogeneity predicate $P(r_{i+1} \cup s_i) =$ true. Two regions s_i, t_i are connected, if they have a common subregion: u_{i-1}. Regions of the lowest level are connected with their neighboring regions. The overlapping structure of the stack of cells guarantees that the clustering process considers only those regions, which are connected or have a path of connected regions on the lowest level. The clustering process is illustrated in Figure 6.

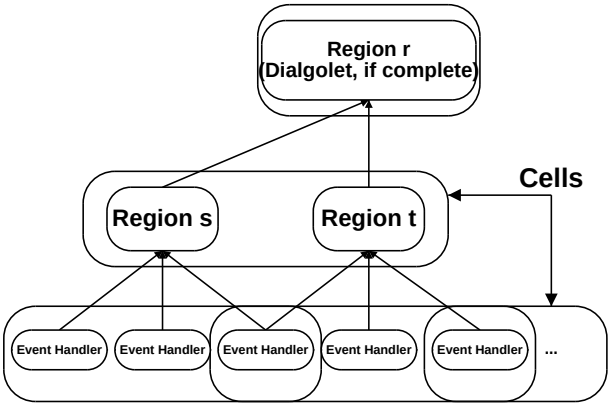


Figure 6: Clustering process. Regions are grouped within the boundary of a cell.

The homogeneity predicate decides, if regions will be clustered or not. The predicate consists of two parts, which both need to evaluate to true; $P(r) = Size(r) \wedge Context(r), r \in R_i$.

- **Size.** On different devices a dialoglet is displayed with a varying number and size of EHs. If the size of a region and its parent region is lower than a predefined threshold (e.g., three times of the screen size) the regions are clustered, otherwise they are separated, either horizontally or top-down. The size of a region is device dependent.
- **Context.** The designer of the original dialog model integrates in it semantic information. The information deals with the semantic relation of an EH with its neighboring EHs. A region s_i and its tentative parent region r_{i+1} will be grouped, if their semantic intent does not exceed a predefined threshold $d(\sigma(s_i), \sigma(r_{i+1})) < \Theta$. In the current version of the adaptation process, we simply assign integer values to EHs, to indicate semantic similarity. $d(\cdot, \cdot)$ is a distance measure like the Euclidian distance.

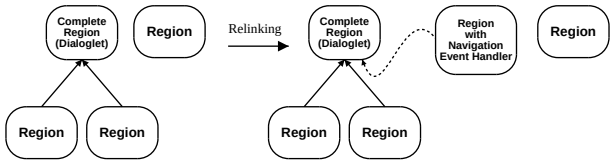


Figure 7: A region containing a single navigation EH will replace a *complete* region. The new region takes part in the bottom-up clustering phase on behalf of the *complete* region.

Top-down Separation: If the grouping process fails, because a region s_i does not satisfy the homogeneity predicate P, the region need to be separated from its connected region t_i. The region need to be separated since they have a common subregion u_{i-1}, which needs to be assigned to a single parent region (Figure 6). The separation process assigns the common subregion to the region, whose semantic value is the most similar. The process is recursively applied down to the lowest level. For level $i - 1$ in Figure 6 it would be applied to u_{i-1}, the common subregion of s_i and t_i, and to those subregions of regions of level i, which have a common subregion with u_{i-1}.

Horizontal Separation: If the size of a region r_i prevents it from clustering with other regions, although it could from the homogeneity predicate's point of view, it is split into a sequence of n smaller, mutually linked regions $r_{0,i}, r_{1,i}, \ldots, r_{n,i}$. E.g., a lengthy text message is split into a sequence of regions or EHs containing each a part of the text message. Only the head of the sequence continues to take part in the grouping process.

Relinking: A region that cannot further be clustered with other regions into a region of a higher level is called *complete* and results in a dialoglet after the adaptation process. A *complete* region that has reached the threshold of the maximal allowed size or that cannot further be clustered from a semantic context point of view does not drop out of the grouping process. Instead, a new region is created containing a single navigation EH pointing to the *complete* region. The new region takes the place of the *complete* region and continues the grouping process on behalf of it. The process is illustrated in Figure 7. The effect of the relinking phase is that the adaptation process creates a linked tree-structure. The regions representing the leaves of that tree-

structure contain the EHs of the original dialog. The intermediate nodes of the tree-structure are regions including the navigation EHs that have been created in the relinking process.

The set of complete regions resulting from the adaptation process are transformed into a set of dialoglets and eventually into a concrete UI applying the Presentation Integrator SPU and the Encoding Transformer SPU of MUSA.

RESULTS

To illustrate the adaptation technique of a dialog model, we have implemented a message board service build with software agents for a location-based systems [7]. The message board contains location specific information and users can read and store messages on the message board. A mobile user moving from location to location accesses different message boards depending on the geographical position. Different users use different devices to access the message board such as laptops, PDAs, or mobile phones. The dialog model

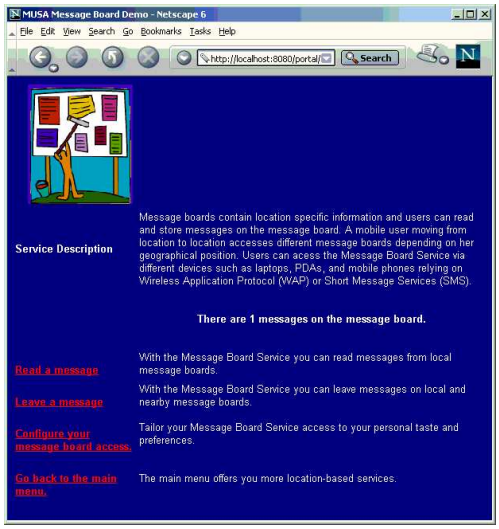


Figure 8: HTML browser showing the Message Board ”Main Menu”.

that results in the graphical UI on a HTML browser is shown in Figure 8. It shows the UI of the message board service. This browser is a powerful tool, so that there is no need to perform any adaptation of the dialog model. Additionally to the dialog a global presentation model is defined, which is responsible for the layout of the Web-page. The two aspects that guide the adaptation process are size and context. The context information is inserted into the dialog model at design time. Size, however, or the screen space that is available for presentation of the UI, is device dependent. The adaptation process needs size information of the device’s UI that accesses the service. This information is delivered in device profiles.

Figure 9 shows the same dialog model that results in the UI of a HTML browser in Figure 8, but this time adapted to the small screen of a mobile telephone. There are two things to note. First, the menu is hierarchically structured into a

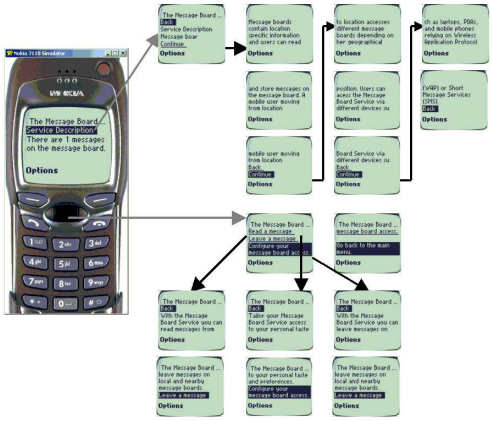


Figure 9: WML-Browser showing the Message Board ”Main Menu” on a mobile telephone.

two level menu, with a main menu containing links to each menu item, which are presented on their distinct screen. The main menu is created during the relinking process of the adaptation and is not present in the original dialog model. The clustering process groups the newly created navigation UI elements together, which results in the main menu. Second, the service description, which is a lengthy text, is split into a series of screens, which are linked with each other. The user navigates with the ”Continue” and ”Back” links from one screen containing part of the description to the next screen. The size threshold of the homogeneity predicate for this adaptation process is set to three device screen sizes.

Figure 10 shows the results for the device profile with the size threshold set to two device screens. The adaptation process has added another level of indirection. The main menu has a hierarchical structure of depth three to cope with the small screen size. The figure shows only part of the collection of UI screens. It illustrates the different hierarchical menu structure in comparison to Figure 9.

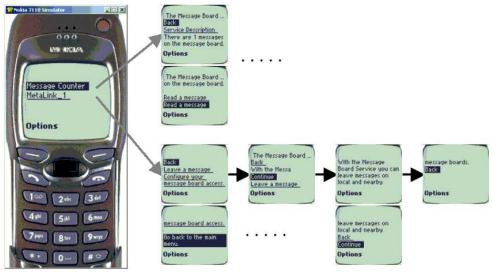


Figure 10: WML-Browser showing the Message Board ”Main Menu” on a mobile telephone with size threshold of two device screens.

CONCLUDING REMARKS

The article has presented the MUSA project. It is a novel approach to device independent authoring. Adaptation of a dialog model represented by the EH graph in EGXML is based on bottom-up clustering and top-down separation us-

ing low-level semantic context information. It results in a hierarchical structure of dialoglets by clustering, separating, and relinking regions and EH. The process is guided by low-level semantic information that is provided by the designer of the dialog model at design time. The adaptation process remodels dynamically a presentation of the dialog model to better fit it to the current device.

The presented experiments with the dialog model adaptation technique are promising and show that the concept is sound. The use of the hierarchy of graph has been proven flexible and is a viable concept for future UI development.

In our future work, we will elaborate the adaptation algorithm to include user specific settings such as window size of the running application or user-preferred font size. We conduct experiments with more complex dialog models. The integration of task model related information into the dialog model is somehow simple. Exploration of more powerful but equally simple methods needs to be carried out. However, simplicity for the designer is an important objective to encourage use of this design technique.

REFERENCES

1. M. Abrams, C. Phanouriou, A. Batongbacal, S. Williams, and J. Shuster. UIML: An Appliance-Independent XML User Interface Language. *WWW8 / Computer Networks*, 31(11-16):1695–1708, 1999.

2. Mark Alpert. Machine Chic. *Sci.Am*, August 2002.

3. D. Atkins, T. Ball, G. Bruns, and K. Cox. Mawl: A domain specific language for form-based services. *IEEE Transaction on Software Engineering*, 25(3):334–346, 1999.

4. T. Ball, P. Danielson, L. Jagadeesan, R. Jagadeesan, K. Laeufer, P. Mataga, and K. Rehor. Sisl: Several Interfaces, Single Logic. *"International Journal of Speech Technology"*, 3:93–108, June 2000.

5. Christian Elting, Jan Zwickel, and Rainer Malaka. Device-Dependant Modality Selection for User Interfaces – An Emprical Study. In *ACM IUI*, San Fransisco, California, USA, January 2002.

6. Mir Farooq and Marc Abrams. Simplifying Construction of Multi-Platform User Interface Using UIML. In *UIML Europe Conference*, "March" 2001.

7. Sebastian Fischmeister. Mobile software agents for location-based systems. In *Agents and Software Engineering*, volume 2592 of *LNCS*, pages 226 – 239. Springer Verlag Heidelberg, 2003.

8. M. Florins and J. Vanderdonckt. Graceful Degradation of User Interfaces as a Design Method for Multiplatform Systems. In *Conference on Intelligent User Interfaces (IUI04)*, pages 140 – 147, Funchal, Portugal, 2004.

9. Futuremind. The Next Generation in Light and Sound Technology Transforms your PC into the Ultimate Mind Machine, 2002.

10. Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August/September 1988.

11. Aaron Marcus and Eugene Chen. Designing the PDA of the Future. *Interactions*, 9(1):34–44, 2002.

12. Guido Menkhaus and Sebastian Fischmeister. Dialog Model Clustering for User Interface Adaptation. In *Web Engineering, Proceedings of ICWE 03*, pages 194 – 203. 2003.

13. Brad Myers. User Interface Software Tools. *ACM Transaction on Computer-Human Interaction*, 2(1):64–103, March 1995.

14. Peter Nacken. Image Segmentation By Connectivity Preserving Relinking in Hierarchical Graph Structures. *Pattern Recognition*, 28(6):907–920, 1995.

15. Paulo Pinheiro da Silva. User Interface Declarative Models and Development Environments: A Survey. In *Proceedings of DSV-IS2000*, pages 207–226, Limerick, Ireland, June 2000.

16. Nikhil R.Pal and Sankar K.Pal. A Review on Image Segmentation Techniques. *Pattern Recognition*, 26(9):1277–1294, 1993.

17. Subhasis Saha, Mark Jamtgaard, and John Villasenor. Bringing the Wireless Internet to Mobile Devices. *IEEE Computer*, 34(6):54–58, 2001.

18. M. Satyanarayanan. Pervasive Computing: Vision and Challenges. *IEEE Persoanl Communications*, pages 10–17, August 2001.

19. Ben Shneiderman. Direct Manipulation for Comprehensible, Predictable and Controllable User Interfaces. In *Intelligent User Interfaces*, pages 33–39, 1997.

20. P. Szekely. Retrospective and Challenges for Model-Based Interface Development. In F. Bodart and J. Vanderdonckt, editors, *Design, Specification and Verification of Interactive Systems '96*, pages 1–27, Wien, 1996. Springer-Verlag.

21. W3C. Mobility Access Activity Statement, 2001.

22. C. Wong, H.H. Chu, and Katagiri M. A. Single-Authoring Technique for Building Device-Independent Presentations. In *W3C Workshop on Device Independent Authoring Techniques*, 2002.

23. UIMS Tool Developers Workshop. A Metamodel for the Runtime Architecture of an Interactive System. *SIGCHI Bulletin*, 24(1):32–37, 1994.

24. Xybernaut. poma, 2002.

Best of both worlds - linking of XUL to X3USGP

Andreas Müller
Chair of Software Engineering
Institute of Computer Science
Albert-Einstein-Str. 21
18059 Rostock, Germany
xray@informatik.uni-rostock.de

Peter Forbrig
Chair of Software Engineering
Institute of Computer Science
Albert-Einstein-Str. 21
18059 Rostock, Germany
pforbrig@informatik.uni-rostock.de

ABSTRACT
In this paper we discuss an approach linking XUL specifications to our own X3USGP (XML-based User Interface Specification and Generation Process, 3 languages) concept. This linking is intended to combine the main advantages of both approaches. On the one hand, it will be presented how the real device independence of the X3USGP concept and the easy handling concrete specifications by XUL can be combined. On the other hand, it will be shown that the disadvantages of both solutions: the restriction to virtual GUI's of the XUL approach and the complicacy of specification of the interaction behavior on a very abstract level of the X3USGP concept can be eliminated.

The main focus is on the device-independent result of the specification of user interfaces and on a simple "human-like" handling of the specification process. Furthermore, a transparent generation process to the resulting user interface without the necessity of proprietary generation tools for each possible target-device is supported.

Keywords
User Interface, XML, Markup Language, XUL,

INTRODUCTION
The presentation of the project CanonSketch [6] during the workshop of "Making Model-Based UI Design Practical: Usable and Open Methods and Tools" inspired us to combine our abstract development approach of user interfaces X3USGP with our more concrete project of a graphical editor GUIXUL [13] for XUL [1,2] specifications within the eclipse [12] environment.

CanonSketch is a graphical editor for describing user interfaces in an abstract way. Such an editor delivers the abstract representation of a user interface, which is the input for our X3USGP approach, which will be discussed in a little bit more detail later on.

However, by looking at the editor of CanonSketch we got the impression that it is very hard for a user interface designer to express his idea immediately in an abstract way. Especially the graphical appearance of abstract element seems to be crucial. It might be easier to design a concrete user interface first and abstract later on to an abstract specification than the other way round. This impression inspired us to combine our ongoing work on a graphical editor for XUL with an abstraction process, which leads to the abstract specification of a user interface. The process is similar to a reverse engineering and a following forward engineering process in software engineering. The starting points are concrete designs, which are abstracted and later on refined to new or alternative designs.

RELATED WORK

UIML
The development of UIML [3] started in 1997. The goal of this project was the specification of device-independent UI's. UIML is based on XML for specification. To generate a specific user interface a generation-tool called renderer is needed. For each possible end-device-type a specific renderer is needed. A flexible reaction to new device-types is difficult.

UIML has a commercial background by the company "Harmonia".

XIML
Another commercial solution ("Redwhale") is the XIML-concept [4,5]. XIML started in 1999 and is focused on device-independence primarily of mobile devices. XIML is model-based but like the UIML-variant it needs a specific tool (converter) to create a specific type of user interface.

CanonSketch
The project CanonSketch [6] is intended to support the requirement analysis process. It is a new type of editor. It looks like a graphical editor but it provides elements of the user interface in a very abstract form only. More precisely, it is a tool for the creation, design and editing of Canonical Abstract Prototypes according to [7].

An editor like CanonSketch is able to deliver a model of abstract interaction models, the intended input of our X3USGP development process.

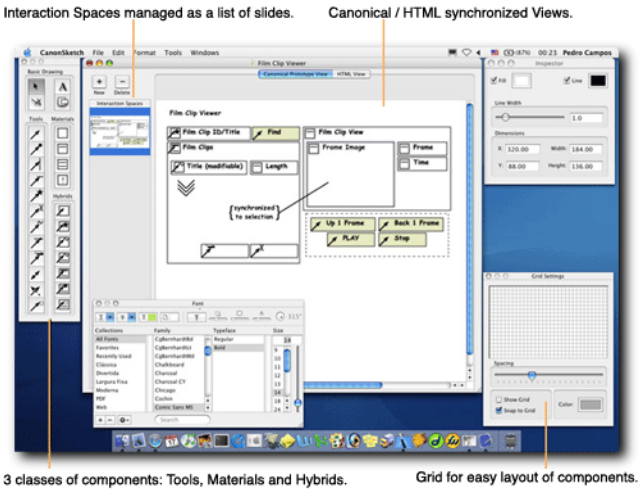


Fig. 1: The CanonSketch editor

THE APPROACHES THE IDEA IS BASED ON

Graphical editor for XUL

XUL [1,2] was presented in 1999 by the mozilla project to specify Graphical User Interfaces of the mozilla-browser in platform-independent matter. XUL allows the specification of interactive objects like buttons, labels, and text fields. Unfortunately, it is restricted to Graphical User Interfaces. Only virtual interfaces can be specified, physical interfaces of information terminals, banking terminals, different mobile phone solutions and other proprietary interface types are not intended to be used for XUL specifications. Thus, there is not really a device independence. Nevertheless, XUL provides the opportunity to specify concrete user interfaces, which can be used as the starting point for abstractions.

Based on an existing open source project v4All [11] a GUI editor was developed for the eclipse environment that stores its results as XUL specifications.

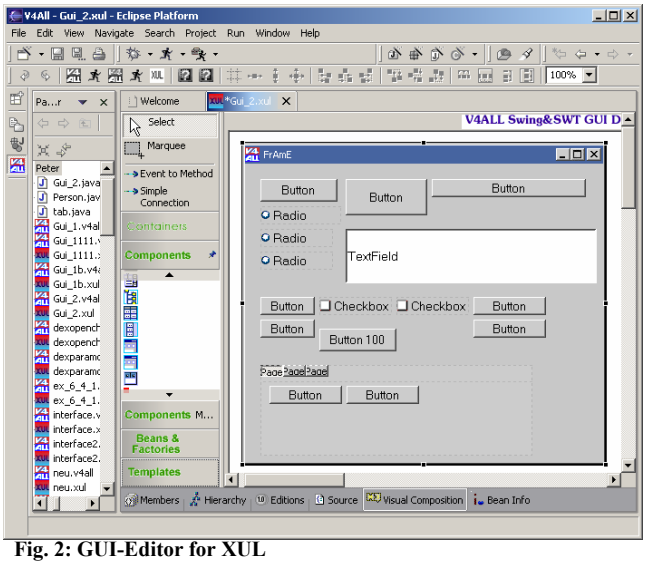


Fig. 2: GUI-Editor for XUL

The editor allows the design of new interfaces. Additionally, it reads an existing XUL specification, displays the graphical user interface, and allows changes to it.

The x3usgp approach

The X3USGP – concept is a multi-stage process that is intended to specify and generate user interfaces in a really device-independent matter. We have presented this concept in detail in [8, 9].

The first step of this process is the specification of the interaction behavior of a user interface on a very abstract level. We use a small set of abstract user interface objects like 1-of-n-choice, m-of-n-choice, trigger-objects, string-objects.

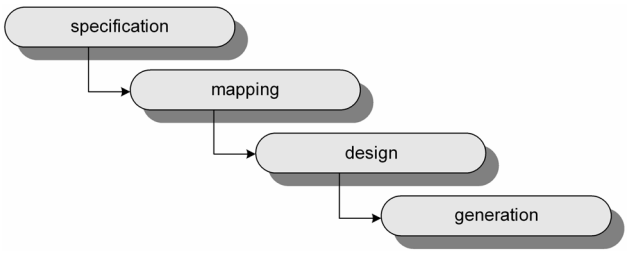


Fig. 3: Multi-stage process

On this abstract level we divide the set of interaction objects into input-objects and output-objects. Later on, we have specific interaction objects that can offer both services and provide input- and output-behavior. The result of this first step is a device-independent specification of the interaction behavior of a user interface – an Abstract Interaction Model (AIM). An XML-based language called X-AIM was presented in [9] for this purpose.

In the second step follows a mapping of this AIM to specific target-devices. Therefore, we need a description of all properties of a specific device, which are relevant for the presentation. Such properties are e.g. color, coordinates, dimension, and label. For each potential target device such a device specification of all design properties is needed. An XML-based language called X-DES [9] was presented for such device specifications.

The result of this second step is a device-dependent specification of interaction behavior – a Specific Interaction Model (SIM). All presentation properties of a specific device are already included in this model. However, at this stage they have no content (value).

At this development stage having, a device dependent specific interaction model, there is only a skeleton of a user interface specified. The design (specifying the design properties by values) will be discussed below. For describing the model, we presented an XML-based language called X-SIM [9].

The mapping procedure described in Fig. 4 is a semiautomatic interactive process. The mapping of an abstract interaction object to a specific one is ambiguous. Some mapping rules are presented in [9] but normally interactive support is necessary.

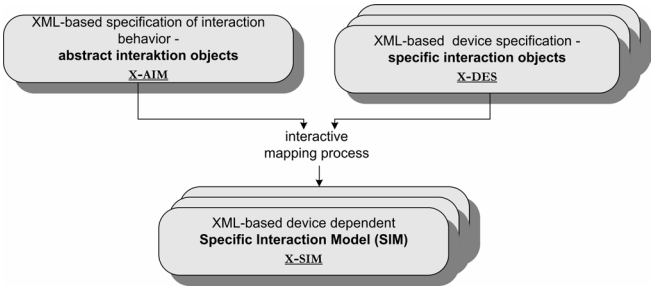


Fig. 4: Mapping process

In the third step the design of a concrete user interface is performed. This is based on the output of the second step that is called SIM.

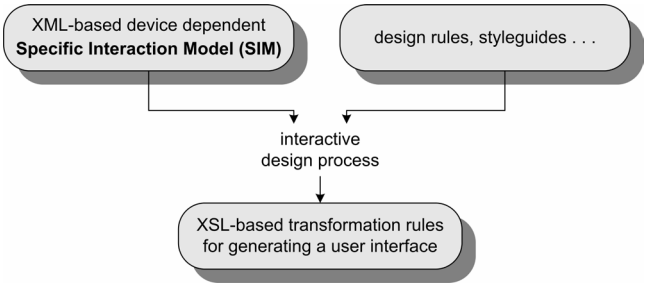


Fig. 5: Design process

This design process is interactive as well. The design properties of a specific device are specified by values. As the result we get XSL-based transformation instructions for generating a concrete user interface.

In the last step a concrete user interface will be created. For this we need an XSL-processor. As the result we get a file representing the user interface (e.g. java, wml, html, a control file for physical devices, etc.)

The specification of interaction behavior takes place in a very abstract matter. This could be a disadvantage of this approach.

LINKING XUL TO X3USGP

Within this paragraph we discuss in a little bit more detail our idea of the development process of user interfaces by:

- Specifying a specific GUI.
- Creating a device-independent abstraction of the GUI.
- Mapping the abstraction of the GUI to other devices.

We will comment on each activity.

Specifying a specific GUI

In contrast to the X3USGP-approach the specification of interaction behavior will not be realized on an abstract level any more. We realized that it is very difficult to discuss with customers the design of abstract user interfaces. The discussion is mostly related to concrete appearances of user interface elements.

We strongly believe that it is often much easier to discuss concrete appearances and later on abstract to more general concepts than the other way round. In our projects sketches on paper with concrete appearances had been made before abstract specifications were developed. Why should a designer develop a user interface and at the same time abstract from concrete representations? The design process is already difficult without abstractions. The abstraction process has to be supported by tools and later on alternative design decisions should be presented.

We suggest to use a GUI editor during the first step of the specification of a user interface as a **graphical user interface**. In our case we suggest the usage of our editor [13] delivering XUL specifications. XUL has a manageable and known set of graphical user interface objects like button, group box, radio box, and slider. We can find these objects in tools for creating GUI's like Java.AWT, Java.Swing and Delphi. This set of user interface objects is platform independent but it is not device independent [9]. There exist also already some tools supporting the interpretation of XUL specifications. Fig. 6 demonstrates the visualization of the GUI specified using our editor (Fig. 2).

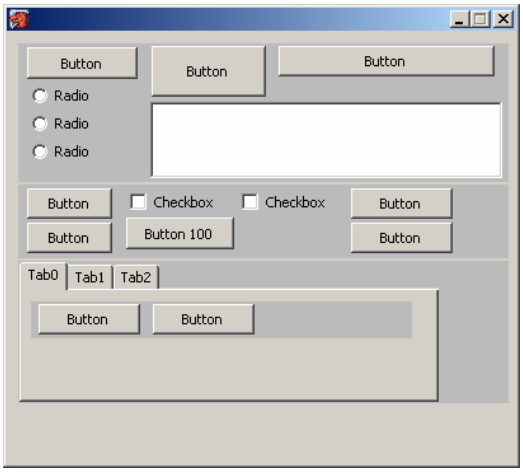


Fig. 6: Interpretation of a XUL specification by the mozilla browser

It was a design decision to use XUL but in principle our approach is not restricted to XUL. XUL can easily be replaced by another language.

Creating of a device-independent abstraction

All objects of the defined set of interface objects in XUL can be mapped to a more abstract one of the Abstract Interaction Model (AIM). To do that we propose **an extension** to XUL, this consists of a table of proper matches of specific objects of XUL to objects of the AIM. This table can be used to support the mapping process from XUL to AIM.

After specification of a prototype interface (dependent to a class of devices: GUI's) we get a **device-independent** result of this XUL-extension, which contains an interface description also with abstract counterparts of user interface

objects. For describing this we develop a XML-based solution. This XML-based result has to contain

- A chosen concrete object in XUL.
- Design properties of this object
- Properties, which specify the behavior or appearance of an object in detail (domain, range, interval etc.).
- The abstract counterpart.

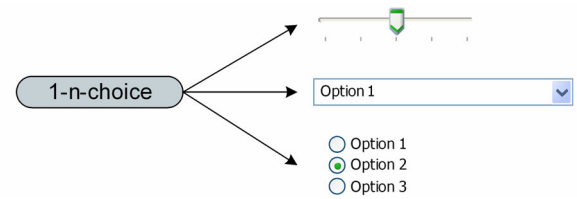


Fig. 7: Abstract object for specific GUI elements

Figure 7 gives an impression how concrete user interface elements can be abstracted to more general elements. The general approach is visualized by figure 8.

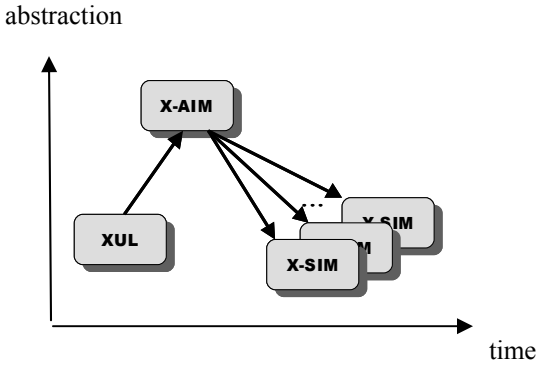


Fig. 8: Developed models

Mapping to other devices

The mapping process of the user interface to different devices depends on the class the device belongs to. There are some advantages if the device belongs to the same class as the original one. In our case it is the class of GUIs.

Devices of the same device class

If we have to map a X-AIM specification to another device of the **same device class** most design properties have already been specified during the first phase – the XUL specification. Indeed, it is easy to create a prototype user interface of another GUI type. We have platform independence within this device class. Probably a manual correction of the resulting prototype is necessary.

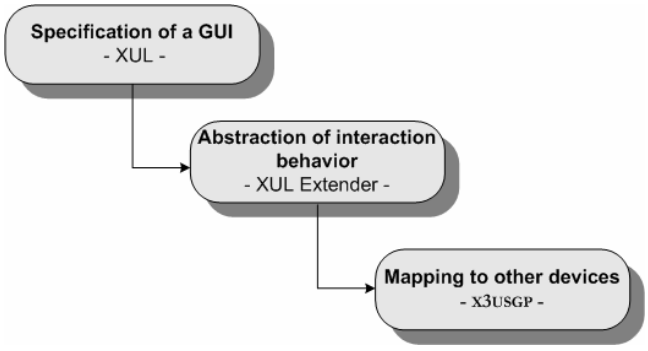


Fig. 9: Development process

Devices of other device classes

For the mapping to a device of another device class (mobile phone, physical info terminal, PDA, etc.), we only need the abstract counterparts of the XUL-extender result. Probably, we can use also other specified properties like labels, domains, or ranges,. Most of design properties like dimensions, and positions can be ignored.

In some sense, this new approach is contrary to the straight development from abstract models to concrete once.

RELATED TOOL SUPPORT

Tool support for the XUL specification step

To support the XUL specification process we presented a GUI for XUL specifications (Fig. 2), which is based on the editor V4All [11].

Tool support for the abstraction process

Tool support for the abstraction support is under development. At the moment transformations have to be made interactively with XSL.

Tool support for the x3usgp process

To support the X3USGP-process we developed an open source prototype solution [9,10]. Currently there is only a tool support for the mapping process. The tool manages different specifications of different devices, reads AIM specifications and creates different SIM specifications in an interactive way.

CONCLUSION

The combination of XUL and the X3USGP – approach seems to be a practicable way to specify device independent user interfaces. It combines the advantages of both solutions – the simple and "human-like" specification of GUI-based user interfaces with given tool support by XUL and the real device-independence of the X3USGP – approach.

A tool support for linking XUL to X3USGP is currently under development. We hope that we are able to

demonstrate the tool with some examples during the workshop

REFERENCES

1. Deakin, N.: XUL Tutorial. XUL Planet. 2000.
2. Mozilla.org: XUL Programmer's Reference 2001
3. Phanouriou, C. and et al: UIML-Specification. 2000.
4. Puerta, Angel and Eisenstein, Jacob: XIML: A Universal Language for User Interfaces. 2001.
5. Puerta, Angel and Eisenstein, Jacob: XIML: A Common Representation for Interaction Data. Poster Reception, IUI2002. 2002.
6. CanonSketch: http://dme2.uma.pt/canonsketch/
7. Constantine, L. Canonical Abstract Prototypes for Abstract Visual and Interaction Design. DSV-IS, Madeira, June 2003, Portugal.
8. Müller, A. and Forbrig, P. and Cap, C.: Model-based User Interface Design Using Markup Concepts. DSVIS2001, Glasgow, 2001
9. Müller, A.: Spezifikation geräteunabhängiger Benutzerschnittstellen durch Markup-Konzepte. PhD-Thesis, University of Rostock. 2003.
10. Hecking, J.-P.: Implementierung des UI-Mapper. University of Rostock. 2001.
11. v4All: http://v4all.sourceforge.net/index_start.html
12. eclipse: http://www.eclipse.org
13. GUIXUL: GUI editor based on [11]