7 HUMAN-CENTERED ENGINEERING OF INTERACTIVE SYSTEMS WITH THE USER INTERFACE MARKUP LANGUAGE

James Helms[1], Robbie Schaefer[2], Kris Luyten[3], Jo Vermeulen[3], Marc Abrams[1,4], Adrien Coyette[5], and Jean Vanderdonckt[5]

[1]Harmonia Inc. P.O. Box 11282 – Blacksburg, VA 24062-1282, U.S.A.
[2]Paderborn University, Fakultät für Elektrotechnik, Informatik und Mathematik, Institut für Informatik, Fürstenallee, 11 – D-33102 Paderborn Germany
[3]Hasselt University - tUL - IBBT Expertise Centre for Digital Media Wetenschapspark 2, 3590 Diepenbeek, Belgium
[4]Dept. of Computer Science, Virginia Tech, 508 McBryde Hall – Blacksburg, VA 24061-0106, U.S.A.
[5]Belgian Lab. of Computer-Human Interaction (BCHI), Louvain School of Management (LSM), Université catholique de Louvain Place des Doyens, 1 – B-1348 Louvain-la-Neuve, Belgium.

Abstract The User Interface Markup Language (UIML) is a User Interface Description Language aimed at producing multiple user interfaces from a single model for multiple contexts of use, in particular multiple computing platforms, thus addressing the need for multi-channel user interfaces. This chapter summarizes efforts devoted to the definition and usage of UIML 4.0, the latest version of this UIDL which also

covers dialog modeling. It describes the main parts of the UIML language, i.e. structure, presentation style, contents, behavior, connectivity, and toolkit mappings, and the integrated development environment that supports the development life cycle of multi-channel user interfaces based on UIML.

7.1 INTRODUCTION

User Interface Description Languages (UIDLs) can bridge the gap between formal Human-Computer Interaction (HCI) models (e.g., with graphical representations) and concrete implementations of User Interfaces (UIs) (Abrams *et al*. 1999). As such, they are an integral part of human-centered engineering of interactive systems and can be applied at any stage of a human-centered design process (Hartson and Hix 1989). In this chapter, we will give an overview on the User Interface Markup Language (UIML) and show, how the fundamental language concepts, tools and extensions as well as the new features of UIML 4.0 can be used to improve a human-centered software development process.

7.1.1 What are UIDLs?

In general, any UIDL should fulfill a series of requirements such as: the ability to specify any target UI, the ability to process a UI specified in the terms of this UIDL, legibility of the resulting specifications, etc. Specifying a UI in a particular UIDL should not be viewed as just an exercise in specification. A UI should be specified as rigorously as possible for multiple purposes: capturing user requirements and turning them into a concrete UI, determining the attributes of a particular UI, describing a UI unambiguously so that this description could be passed to the developers, whether they use a software tool or not. A UIDL could be interpreted as a concrete syntax for UI specification similar to specification languages for the other areas of computer science (e.g., for data base management systems). As such, it could take any form. Expressing this concrete syntax in an XML format is nowadays particularly popular since XML-based technology has received enough attention to be widely used and effectively supported by appropriate software.

7.1.2 UIML and Software Engineering

In this chapter we focus on UIML, an open, standardized XML language to express 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 organizations. Using UIML, HCI best practices can be adopted more easily by viewing HCI design and implementation as a process of transformation from initial design model to an open standard XML UI implementation language and finally to target languages for deployment. UIML defines mappings between the abstractions used in the UI description and the underlying implementation, and UIML implementations make liberal use of transforms to convert from one

set of abstractions to another. Note that the target languages can be XML languages (e.g., XHTML, SVG) or programming languages (e.g., C# and Java).

There are many XML-based UIDLs. In addition to languages from W3C (HTML, XHTML, XForms, and SVG), early efforts include UIML, the Alternate Abstract Interface Markup Language (Zimmermann et al., 2002), the eXtensible Interface Markup Language (XIML) (Puerta and Eisenstein, 2002) and the XML User Interface Language (XUL) (`www.mozilla.org/projects/xul/`). More recently we have seen the onset of DISL (7.3.4) and UsiXML.

One challenge in creating UIDLs is designing the language to support the software engineering aspects of HCI engineering over 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), and for any UI metaphor.

We will explore how UIML supports software engineering by providing the following benefits:

- Reusability of HCI design components and modules (see section 7.2)
- Abstraction and domain-specific vocabularies to allow generalized use of the language(see section 7.2.3)
- Adaptability to implementation languages and display devices (see section 7.3.1)
- Separation of HCI engineering concerns (see section 7.2.1)
- Rapid Prototyping for user-centered design (see section 7.3.1)
- Tool support (see section 7.3)
- A unified language for representing UIs

In this chapter, we will show how the fundamental UIML concepts, tools and extensions, as well as the new features of UIML 4.0 can be used to improve a human-centered HCI development process.

7.2 UIML: AN OVERVIEW

The User Interface Markup Language (UIML) is a declarative, XML-compliant metalanguage for describing UIs. The design objective of the UIML is to provide a vendor-neutral, platform independent, canonical representation of any UI suitable for mapping (rendering) to existing implementation languages, and as such an ideal candidate for human-centered engineering of interactive systems. Work on UIML was motivated by the following goals:

- Allow individuals to implement UIs for any device without learning languages and application programming interfaces (APIs) specific to the device.
- Reduce the time to develop UIs for a family of devices.
- Provide a natural separation between UI code and application logic code.
- Allow non-programmers to implement UIs.
- Permit rapid prototyping of UIs.
- Simplify internationalization and localization.
- Allow efficient download of UIs over networks to client machines.
- Allow extension to support UI technologies that are invented in the future.

UIML provides a puzzle piece to be used in conjunction with other technologies, including HCI design methodologies, modeling languages such as XIML (Eisenstein, Puerta and Vanderdonckt 2001) and UsiXML (Limbourg and Vanderdonckt, 2004b), authoring tools, transformation algorithms, and existing languages and standards (W3C and OASIS specifications). UIML is not a silver bullet that replaces human decisions needed to create UIs.

UIML is biased toward an object-oriented view and toward complementing other specifications (e.g., SOAP, XForms Models, XHTML, HTML, WML, VoiceXML). During the design of UIML an effort was made to allow interface descriptions in UIML to be mapped with equal efficiency to various vendor's technologies (e.g., UIML should efficiently map to both ASP .Net and JSP to further the goal of vendor-independence in Web Services).

Why is a canonical representation useful? Today, HCIs are built using a variety of languages: XML variants (e.g., HTML, XHTML, VoiceXML,), JavaScript, Java, C++, etc. Each language differs in its syntax and its abstractions. For example, the syntax in HTML 4.0 to represent a button is <button>, and in Java Swing "JButton b = new JButton;". The work on UIML asks the fundamental question, "Do we inherently need different syntaxes, or can one common syntax be used?" The benefit of using a single syntax is analogous to the benefit of XML: Software tools can be created for a single syntax (UIML), yet process UIs destined for any existing language. For example, a tool to author UIs can store the design in UIML, and then map UIML to target languages in use today (e.g., HTML, Java) or invented in the future. Progress in the field of UI design can move faster, because everyone can build tools that either map interface designs into UIML or map UIML out to existing languages. Tools can then be snapped together using UIML as a standard interchange language.

There is a second benefit of a canonical UI description. By using a single syntax to represent any UI, an interface is in a very malleable form. For example, one technique gaining popularity in the human computer interface community is transformation. With a canonical representation for any UI, someone that designs a transform algorithm can simply implement the algorithm to transform an input UIML document

to a new output UIML document. Compare this approach to implementing the same transform algorithm only to transform HTML documents, then reimplementing the transform algorithm to only transform C++ interfaces, and so on.

In any language design, there is a fundamental tradeoff between creating something general versus special-purpose. UIML is for general-purpose use by people that implement UIs and people that build tools for authoring UIs. It is envisioned that UIML will be used with other languages with a more focused purpose, such as UI design languages. Ultimately most people may never write UIML directly – they may instead use a particular design language suited to a certain design methodology, and then use tools to transform the design into a UIML representation that is then mapped to various XML or programming languages.

UIML is object-based, in that it describes both objects, classes, and the interactions amongst objects. UIML was designed to allow UI descriptions to be mapped with equal efficiency to different vendors' technologies. Re-use is a first class concept in UIML and is accomplished through a template mechanism. UIML templates can contain snippets of UIML that can then be inserted into other UIML documents. This capability allows designers to build up libraries of user interface components and style sheets that can be applied throughout their systems. UI style guidance can be directly enforced via this mechanism. Libraries defined in this way allow custom components to be defined once and reused, which can be very helpful for user-centered software engineering e.g. by adhering to usability patterns. Four key concepts underlie UIML:

1. *UIML is a meta-language.* To understand this, consider XML. XML does not define tags, such as <p>. Instead, one must add to XML a specification of the legal tags and their attributes, for example by creating a document type definition (DTD). Therefore the XML specification does not need to be modified as new tag sets are created, and a set of tools can be created to process XML independent of the tag sets that are used. UIML defines a small set of powerful tags, such as <part> to describe a part of a UI, or <property> to describe a property of a UI part. UIML tags are independent of any UI metaphor (e.g., graphical UIs), target platform (e.g., PC, phone), or target language to which UIML will be mapped (e.g., Java, HTML). To use UIML, one must add a toolkit vocabulary (roughly analogous to adding a DTD to an XML document). The vocabulary specifies a set of classes of parts, and properties of the classes. Different groups of people can define different vocabularies, depending on their needs. One group might define a vocabulary whose classes have a 1-to-1 correspondence to HCI widgets in a particular target language (i.e., the classes might match those in the Java Swing API). Another group might define a vocabulary whose classes match abstractions used by a UI designer (e.g., Title, Abstract, BodyText for UIs to documents). UIML is eventually standardized once and tools can be developed for UIML, independently from the development of vocabularies. For more on vocabularies see section 7.2.3.

2. *UIML "factors out" or separates the elements of a UI.* The design of UIML started with the question: what are the fundamental elements needed to describe any man-machine interaction. The separation in UIML identifies what parts

comprise the UI, the presentation style for each part as a list of <property> elements, the content of each part (e.g., text, sounds, images) and binding of content to external resources (e.g., XML resources, or method calls in external objects), the behavior of parts when a user interacts with the interface as a set of rules with conditions and actions, the connection of the UI to the outside world (e.g., to business logic), and the definition of the vocabulary of part classes. For a comparison of the separation in UIML to existing UI models, such as the Model View Controller, refer to Phanouriou (2000).

3. *UIML views the structure of a HCI, logically, as a tree of UI parts that changes over the lifetime of the interface*. There is an initial tree of parts, which is the UI initially presented to a user when the interface starts its lifetime. During the lifetime of the interface, the tree of parts may dynamically change shape by adding or deleting parts. For example, opening a new window containing buttons and labels in a graphical interface may correspond to adding a sub-tree of parts to the UIML tree. UIML provides elements to describe the initial tree structure (<structure>) and to dynamically modify the structure (<restructure>).

4. *UIML allows UI parts and part-trees to be packaged in templates.* Templates may then be reused in various interface designs. This provides a first class notion of reuse within UIML, which is missing from other XML HCI languages, such as HTML.

Due to these concepts, UIML is particularly useful for creating multiplatform UIs and also personalized solutions for different users. To create multiplatform UIs, one leverages the meta-language nature of UIML to create a vocabulary of part classes (e.g., defining class Button), and then separately defines the vocabulary by specifying a mapping of the classes to target languages (e.g., mapping UIML part class Button to class javax. swing.JButton for Java and to tag <button> for HTML 4.0). One can create a highly device-independent UI by creating a generic vocabulary that tries to eliminate bias toward particular UI metaphors and devices. By "device" we mean PCs, various information appliances (e.g., handheld computers, desktop phones, cellular or PCS phones), or any other machine with which a human can interact. In addition, because UIML describes the interface behavior as rules whose actions are applied to parts, the rules can be mapped to code in the target languages (e.g., to lines of Java code or JavaScript code).

UIML is currently being standardized in the Organization for the Advancement of Structured Information Standards (OASIS, `www.oasis-open.org`) by a technical committee comprised of UI definition language experts from across the globe. The technical committee's web page can be accessed at `www.oasis-open.org/committees/tc_home.php?wg_abbrev=uiml`.

7.2.1 Generalizing the Model View Controller Pattern for UIs

UIML is modeled by the placeMeta-Interface Model (Phanouriou 2000) pictured in Figure 7.1.

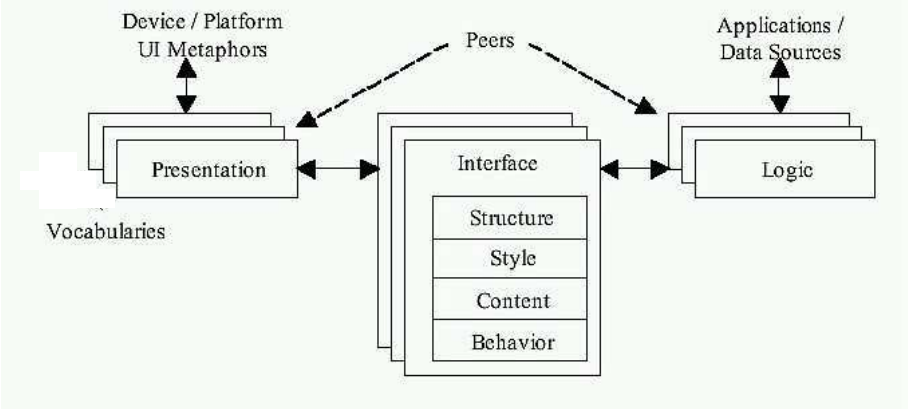

Figure 7.1 The Meta-Interface Model

UIML is designed to describe arbitrarily complex UIs on any device through a unique division of the UI definition into the following six elemental components:

1. *Structure*: the hierarchy of parts in the UI.

2. *Presentation style*: the sensory characteristics of the parts in the UI.

3. *Content*: the words, images, etc. appearing or being spoken in the UI.

4. *Behavior*: a set of condition-action rules that define what happens when a user interacts with an UI element, such as a button.

5. *Connectivity* (mappings to object methods in the application layer).

6. *Toolkit mappings* (mappings of the object class names in the hierarchy of parts to specific objects in the toolkit used to render the UI.

Each of these elements can be described independently to allow re-use and flexibility in creating UIs for widely different platforms. In addition, the separation of concerns inherent to UIML was developed to generalize the Model View Controller design pattern for use with UIs. In UIML, the Interface shown in Figure 7.1, serves as the model of the UI, describing the structure, style, content, and behavior of the HCI in a canonical way. The View is then generated from the presentation which maps concepts used in the interface (model) to concrete widgets on the screen. Finally the Logic acts as the controller bridge between the HCI and the model. This allows the UI to communicate with the application without defining specifics of the communication (protocols, syntax, etc.).

7.2.2 Hello World Example

The famous “Hello World” example in UIML is shown in Figure 7.2. It simply generates a non-interactive UI that contains the words ”Hello World!”. The example illustrates the typical structure of a UIML UI description. Notice how the interface definition is divided into a separate structure and style child. The structure child contains all the parts that define the user interface, in this case a container called “TopHello” and

```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC "-//OASIS//DTD UIML 3.1 Draft//EN"
    "http://uiml.org/dtds/UIML4\_0a.dtd">

<uiml>
  <interface>
    <structure>
      <part id="TopHello">
        <part id="hello" class="helloC"/>
      </part>
    </structure>
    <style>
      <property part-name="TopHello" name="rendering">TopContainer</property>
      <property part-name="TopHello" name="title">Hello</property>
      <property part-class="helloC"  name="rendering">String</property>
      <property part-name="hello"    name="content">Hello World!</property>
    </style>
  </interface>
  <peers> ... </peers>
</uiml>
```

Figure 7.2 The UIML specifications for the "Hello World!" example

a label called "hello". The hierarchical nature of the structure definition implies that "hello" is a child of "TopHello" and is therefore contained within it. The style section enumerates the presentation properties on the two parts that will affect the appearance of the UI. For example, the "TopHello" container will have "Hello" as the title and the label will contain "Hello World!" as specified by the title and content properties, respectively. Properties are associated with parts through the *part-name* or *part-class* attribute. If a property is meant to apply to only one part then we use *part-name* to specify by id the part that will exhibit this property. If *part-class* is used then the property will be applied to all parts of that class.

7.2.3 Generalizing through Vocabularies

The richness of the user interfaces that can be described with a UIDL is proportional with the expressive power of the presentation model. There are two extremes with regard to the expressive power of a UIDL: the *common denominator* and the *meta-widget* set approach. On the one hand the common denominator approach identifies a general set of abstract interactors that can be used on most platforms or devices. This set is used to create the UIDL: the language's syntax is limited to describing user interfaces composed out of elements from the general set. On the other hand, the meta-widget set approach avoids including any widget-set specific information in the language. In this case, the UIDL is a meta-language that describes different aspects of a user interface that are independent of any widget set, platform or device. UIML is such a canonical language that adheres the meta-widget set approach.

The mapping vocabulary is the part of UIML that allows using a custom naming scheme while creating the user interface description. This implies the names used

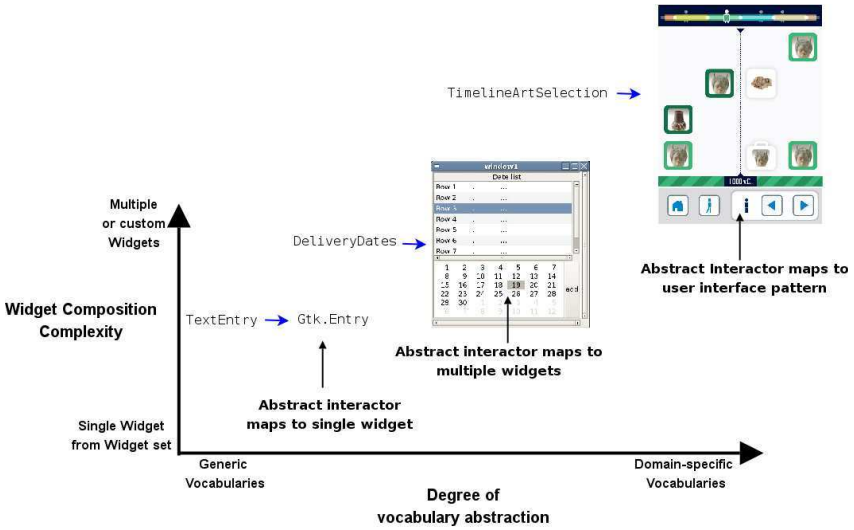

Figure 7.3 Graphical overview of abstraction by using vocabularies.

in UIML to describe the user interface can be domain-specific names and the naming scheme defined in the external vocabulary will map these names onto platform-specific names. For example, a UIML document for in-car navigation systems could use names as "route", "speed", "crossroad" and the vocabulary will tell how to present these domain specific names in a concrete widget set. Similar, a vocabulary for building music player user interfaces could define names as "playlist", "arrangement" and "track". Support for custom naming schemes allow creating vocabularies that abstract the problem domain and uses names known to the domain experts. Figure 7.3 provides a graphical overview of the vocabulary abstraction idea.

The standard UIML vocabulary allows mapping one domain object onto one *widget class* but does not allow mapping a domain object onto *a set* of widget classes or a *user interface pattern*. We are currently extending the mapping vocabulary with more semantics, of which an example is shown in the listing below. The listing shows how a date part could be mapped on different types of concrete widgets, or even on a template containing a user interface pattern. More on templates can be found in section 7.4.3

Vocabularies can be used to tailor UIML to the needs of specific deployment platforms. For example, Harmonia has developed vocabularies for Java, C++/Motif, HTML, WML, and VoiceXML. Harmonia also developed a generic vocabulary that uses abstract component classes and properties to describe visual interfaces (Ali et al., 2002). Such a generic vocabulary allows mappings to be defined from the abstract class and property names to multiple platform specific widget toolkits. This approach enables a single UIML file to be mapped to multiple platforms without rewriting it. To accommodate other user interface toolkits, one only needs to define an appropriate vocabulary.

```
  <uiml:d-class id="Date" used-in-tag="part" maps-type="class">
   <xsl:choose>
    <xsl:when test="expression1">
     <uiml:maps-to name="Gtk:Label">
       <uiml:d-property id="date" maps-type="setMethod" maps-to="Text.Concat">
         <uiml:d-param name="day" type="int"/>
         <uiml:d-param name="month" type="int"/>
         <uiml:d-param name="year" type="int"/>
       </uiml:d-property>
       <uiml:d-property id="selectable" maps-type="setMethod" maps-\\
          to="Selectable">
         <uiml:d-param type="bool"/>
       </uiml:d-property>
       ...
     </uiml:maps-to>
    </xsl/when>
    <xsl:when test="expression2">
     <uiml:maps-to name="Gtk:Calender">
       <uiml:d-property id="date" maps-type="setMethod" maps-to="Date">
         <uiml:d-param name="day" type="int"/>
         <uiml:d-param name="month" type="int"/>
         <uiml:d-param name="year" type="int"/>
     </uiml:d-property>
     <uiml:d-property id="selectDay" maps-type="setMethod" maps-\\
         to="SelectDay">
        <uiml:d-param type="System.Int"/>
     </uiml:d-property>
     ...
   </uiml:maps-to>
  </xsl:when>
   ...
  <xsl:otherwise>
    <uiml:maps-to name="mytemplate" source="\#mytemplate1" how="replace" />
  </xsl:otherwise>
 </xsl:choose>
...
```

Figure 7.4 Listing of mapping of a date part on types of concrete widgets.

7.3 TOOLS FOR AND EXTENSIONS OF UIML

The benefits of a language cannot be realized without a set of tools designed to utilize the language. Therefore a set of commercial and non-commercial tools and applications based on UIML have been produced to serve different developer communities. These tools are all based on the UIML specification and are available for a wide range of markets and different types of end-users: this section describes UIML-based tools that range from high-quality commercial-level products on the one hand to highly experimental and open source software on the other hand.

	Product Name	Capability
Integrate	**uReuse**^{TM}	Analyze legacy resources to expose interface; encapsulate for Flex
	uInventory^{TM}	Create and search module and transform library with metadata
	uGlue^{TM}	Create data transformations
Develop	**uDevelop**®	Design HCIs with library modules, hook up services, link to process
	uRender^{TM}	Implement graphic design of HCI to variety of target devices
	uTest^{TM}	Test automatically using scenario-driven regression testing across platforms
Manage	**uMeasure**^{TM}	Collect metrics on key/mouse movement; analyze human; estimate processing time
	uManage^{TM}	Manage versions; integrate with requirements, configuration. & project management
	uLearn^{TM}	Emit automatically training parts generated from & synchronized to business process, ready for flushing out; emit technical manuals

Table 7.2 Definition of LiquidApps^{TM} Components.

7.3.1 LiquidApps^{TM}

Harmonia's LiquidApps^{TM} product suite is a comprehensive development and deployment environment for the rapid assembly of applications. Using LiquidApps^{TM}, applications are composed from an ecosystem of reusable application components. These components can be custom built, assembled from other more primitive components, or extracted from existing systems. LiquidApps^{TM} utilizes UIML to describe all aspects of an application's interface and connection to the presentation logic. LiquidApps^{TM} began life as LiquidAPPS^{TM}, a UI development environment based solely on UIML. Now, it retains its HCI focus while extending the lessons learned with UIML to the rest of the application development process.

LiquidApps^{TM} serves as a testbed for the application of UIML in multiple facets of the UI design and development lifecycle. The product suite implements a forward-looking vision to provide comprehensive support for all aspects of rapid application development and integration into the software lifecycle, with a focus on the user's HCI experience. LiquidApps^{TM} has been used for UI prototyping in major Defense applications. In addition, LiquidApps^{TM} users include the US Navy, prime US Government contractors, the US Department of Energy, the automotive multimedia interface collaboration (AMI-C, `www.ami-c.org`), and European consumer electronics manufacturer Beko Elektronik. The LiquidApps^{TM} suite consists of the products shown in Table 7.2.

Each product in the LiquidApps^{TM} suite provides a different avenue of exploration that helps to test the limits of UIML's utility and function. The individual tools generally fall into the following three categories:

1. *Integration Products*: products in this category focus on extracting re-usable modules from existing software systems and allowing them to be assembled and re-used in new UIs.
2. *HCI Development Products*: these tools provide mechanisms for creating UIML UI descriptions and synchronizing these descriptions to software engineering design artifacts. Deployment tools then take the UIML descriptions produced by other products and create deployable UIs from them. For example, products in this category could take a UIML file and automatically produce C++ source code for the UI.
3. *Management Products*: products in this category synchronize the UI description to training development, version control, and performance measurement. This helps to ensure that designs produced using the tool adhere to a user-centered methodology and provide the appropriate supplemental functionality to integrate into all aspects of the HCI design and software engineering process.

Trial versions of LiquidApps^{TM} can be requested from `http://www.harmonia-inc.com/products/index.php`.

7.3.2 SketchiXML

Among the various tools which are compliant with UIML, there also exists a sketching tool where a designer is able to sketch a graphical user interface (Figure 7.5) and to export it in UIML in order to automatically generate its code. This software is called SketchiXML (Coyette & Vanderdonckt, 2005) and can be downloaded from `http://www.usixml.org/`. Figure 7.5 reproduces a typical sketching sessions where various UI elements are provided. This sketching tool addresses the following requirements for quickly producing a very first UI prototype.

Indeed, designing "the" right UI is very unlikely to occur the first time, even with experience. Instead, UI design is recognized as a process that is intrinsically open (new considerations may appear at any time), iterative (several cycles are needed to reach an acceptable result), and incomplete (not all required considerations are available at design time). Consequently, means to support early UI design has been extensively researched to identify appropriate techniques such as paper sketching, prototypes, mock-ups, diagrams, etc. Most designers consider hand sketches on paper as one of the most effective ways to represent the first drafts of a future UI. This kind of unconstrained approach presents many advantages: sketches can be drawn during any design stage, it is fast to learn and quick to produce, it lets the sketcher focus on basic structural issues instead of unimportant details (e.g., exact alignment, typography, and colors), it is very appropriate to convey ongoing, unfinished designs, and it encourages creativity, sketches can be performed collaboratively between designers and end-users.

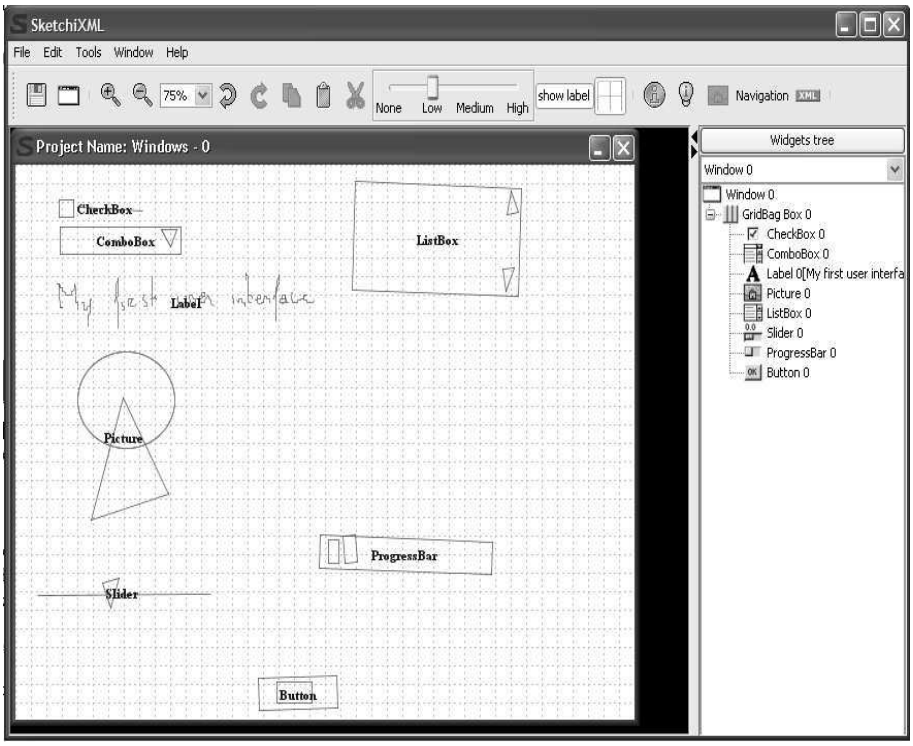

Figure 7.5 A general screen shot of the sketching tool

When the sketch is close enough to the final UI, an agreement can be signed between the designer and the end user, thus facilitating the contract and validation.

The first step in SketchiXML consists of specifying parameters that will drive the fidelity prototyping process: the project name, the input type (i.e. on-line sketching or off-line drawing that is scanned and processed in one step), the computing platform for which the UI is prototyped (a predefined platform can be selected such as mobile phone, PDA, TabletPC, kiosk, ScreenPhone, laptop, desktop, wall screen, or a custom one can be defined in terms of platform model), the output folder, the time when the recognition process is initiated, the intervention mode of the usability advisor (manual, mixed-initiative, automatic), and the output quality stating the response time vs. quality of results of the recognition and usability advisor processes. After that, the designer is free to naturally draw virtually anything on the sketching area. Depending on the parameters, what you sketched is all what you get (the sketch remains drawn as it is, thus preserving the naturalness of its role) or what you sketched is what will be recognized (in this case, a shape recognition engine detects a familiar shape like a widget, an image, a frame, and transforms it into its real counterpart, thus producing more precise specifications). It is not mandatory to sketch only widgets which are

Slider	

Figure 7.6 Various alternative representations of the same widget (here, a slider).

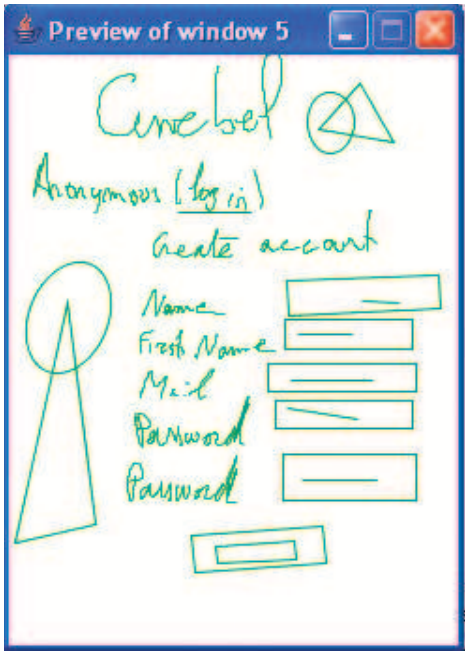

Figure 7.7 Example interface sketch.

recognized: another representation of the same widget could be accommodated or the initial one (Figure 7.6).

7.3.3 UIML.net: a UIML renderer for .NET architecture

UIML.net is a highly dynamic open source UIML rendering engine that interprets and transforms a UIML document into a concrete working user interface. The UIML.net renderer was created to cope with a wide range of devices and to minimize the effort required to support new devices or platforms (Luyten and Coninx, 2004). To accomplish this, vocabularies (or peers) are loaded and processed at run-time by the renderer instead of pre-programmed in the renderer software. The rendering engine queries the vocabulary while rendering the user interface so no widget-set or platform specific information needs to be included in the rendering software itself. Instead, the mappings defined in the vocabulary are interpreted at runtime and loaded on demand. If a vocabulary is changed, a user interface rendered with this vocabulary will be automatically adapted according to the altered mapping rules and presented using the

```
<?xml version="1.0" encoding="UTF-8" ?>
- <uiml>
  - <interface id="interface" how="replace" export="optional">
    - <structure how="replace" export="optional">
      - <part id="GridBag_Box_0" class="JFrame" where="last" how="replace" export="optional">
        - <style how="replace" export="optional">
            <property name="size" how="replace" export="optional">0, 0</property>
            <property name="layout" how="replace" export="optional">null</property>
            <property name="resizable" how="replace" export="optional">true</property>
            <property name="title" how="replace" export="optional">Project Name</property>
          </style>
        - <part id="Label_0" class="JLabel" where="last" how="replace" export="optional">
          - <style how="replace" export="optional">
              <property name="text" how="replace" export="optional">Label 0</property>
              <property name="bounds" how="replace" export="optional">29,23,74,20</property>
            </style>
          </part>
        - <part id="Picture_0" class="JLabel" where="last" how="replace" export="optional">
          - <style how="replace" export="optional">
              <property name="background" how="replace" export="optional">blue</property>
              <property name="opaque" how="replace" export="optional">true</property>
              <property name="bounds" how="replace" export="optional">136,8,41,27</property>
            </style>
          </part>
        + <part id="Label_1" class="JLabel" where="last" how="replace" export="optional">
        + <part id="Label_2" class="JLabel" where="last" how="replace" export="optional">
        + <part id="Picture_1" class="JLabel" where="last" how="replace" export="optional">
        + <part id="Label_3" class="JLabel" where="last" how="replace" export="optional">
        + <part id="Label_4" class="JLabel" where="last" how="replace" export="optional">
        + <part id="Label_5" class="JLabel" where="last" how="replace" export="optional">
        + <part id="Label_6" class="JLabel" where="last" how="replace" export="optional">
        + <part id="Label_7" class="JLabel" where="last" how="replace" export="optional">
        + <part id="Button_0" class="JButton" where="last" how="replace" export="optional">
```

Figure 7.8 An example of a UI for a PDA and its corresponding UIML code.

appropriate concrete widgets. This approach allows for a more sustainable user interface over time: while platforms and devices evolve, a UIML document can be easily reused. However, the UI rendered from the UIML document also takes advantage of the changes.

A prerequisite for the renderer to work is the availability of a virtual machine on the target device. Although the original UIML.net implementation was developed for the Microsoft .Net framework (both the standard framework and the compact framework), there is also a Java implementation based on the .Net implementation. Since most mobile and embedded devices support a Java or .Net virtual machines, UIML.net is almost ubiquitous. Our implementation is built in such a way that there are no further dependencies on exactly which version or type of virtual machine, it simply relies on the virtual machine as a dynamic execution environment that allows loading new functionality on demand while running the user interface. This enables us to support multiple mapping vocabularies on-the-fly.

The architecture of UIML.net consists of a rendering core and multiple rendering backends that contain code that is only used by specific vocabularies. A rendering core can process a UIML document and builds an internal representation of the UIML document. Notice that the mappings from abstract interactors to concrete widgets are defined in the peers section (the vocabulary) of a UIML document. Since the mapping information is provided from outside of the renderer, it can be loaded dynamically and applied at runtime to the rendering core. The rendering backends have a very limited responsibility: they process the parts of a UIML document that can rely on widget-set

specific knowledge. Although this seems to break with the promise of being a ubiquitous rendering engine, the specific rendering backends are not required to create a working user interface. They provide access to platform specific widgets which is often required to create an optimal user experience for a specific platform. The reflection mechanism allows using the information from the vocabulary to detect and load the required widgets at runtime without any platform-specific code. This approach overcomes the limitations of a generic vocabulary by allowing the designer to also use widget-set specific parts.

Once UIML.net has processed the UIML document, an internal representation of the document is build as a set of objects structured as an in-memory tree which can be rendered at any time without further transformations. This internal representation is similar to a Document Object Model tree, and is accessible by an internal API. We define a concrete instantiation of a UIML document as a set of objects that represent the in-memory tree of a UIML document where each object that represents a UIML part has a reference to a final widget on the target platform. A concrete instantiation is the result of executing the mapping rules included in the vocabulary without rendering the user interface on screen. A concrete instantiation can be used to perform post-processing operations that will be discussed in the following paragraphs.

The UIML.net renderer uses several stages, where each stage processes a certain aspect of the UIML document. UIML.net uses three different stages of processing that are required for a flexible ubiquitous rendering engine:

- **pre-processing:** during this stage a UIML document can be transformed into another UIML document. For example; the style properties of the user interface can be changed to avoid using green and red for users suffering from color blindness.
- **main processing:** during this stage a UIML document will be interpreted and a concrete instantiation of the document using the UI toolkits that are available on the target platform will be generated.
- **post-processing:** during this stage the runtime behavior strategies of the UI will be selected. For example; the instantiated layout can be changed to optimize the user experience.

The main processing stage is more specifically composed out of the following steps;

- The UIML-renderer takes an UIML document as input, and looks up the rendering backend library that is referred to in the UIML document.
- An internal representation of the UIML document is built. Every part element of the document is processed to create a tree of abstract interface elements.
- For every part element, its corresponding style is applied.
- For every part element, the corresponding behavior is attached and the required libraries to execute this behavior will be loaded just-in-time.

- The generated tree is handed over to the rendering module: for every part tag, a corresponding concrete widget is loaded according to the mappings defined in the vocabulary and linked with the internal representation. For the generated concrete widget, the related style properties are retrieved, mapped by the vocabulary to concrete widget properties and applied on the concrete widget.

7.3.4 DISL: A Modality Independent UIML-Variant for Limited Devices

Within a project to develop an architecture that allows the provision and management of user interfaces for different devices and modalities - potentially in a multimodal setting - the Dialog and Interface Specification Language (DISL) has been established (Mueller et al., 2004). DISL can be considered as a modified subset of UIML 4.0 and has been designed with the following two major goals in mind:

- The language should be generic enough to be independent of platforms, devices and even modalities.
- The language should be supportive of devices with limited processing and memory capabilities.

UIML itself could be modality independent, as the interface description itself can be done quite generic and the vocabularies are used to map the user interface to specific targets. However, either, a range of vocabularies is needed and the right target platform is selected at runtime as done for example in (Luyten et al. 2006), c.f section 7.3.3, or a multi-stage process to map abstract interfaces to more concrete instances is advisable like the process defined in (Ali et al., 2002).

Anyway both approaches require the definition of several mapping vocabularies. With DISL, the opposite strategy was selected, which means that DISL does not provide the mappings but relies either on an external transcoding process or a dedicated DISL renderer on the target device. So instead of assigning a class to a part which connects to the peers section, a fixed set of generic widgets (inspired by a generic vocabulary for graphical and speech driven applications (Plomp and Mayora-Ibarra 2002) and the concept of abstract interaction objects (Vanderdonckt and Bodart 1993)) can be used with a special attribute of the part. The choice of generic widgets was mainly limited to those allowing input with and without data, output with or without data, confirmations, choices and logical grouping.

In addition, the communication with the backend logic was simplified, so that only calls to a remote or local peer are supported within the action part, e.g. through HTTP or RMI. Again, the communication method relies on the implementation of the renderer and does not have to be modeled separately.

Through these means, the peers section is obsolete, which decreases the language complexity and the size of concrete instances dramatically. However, this was a design decision for DISL and the selected generic widgets could be implemented with a proper vocabulary in UIML just as well.

Even though DISL is as such simpler than UIML, UI descriptions in DISL still consume a lot of memory (at least from the limited device perspective) and parsing a complex tree structure can soon outrun the maximum stack- or heap size of the underlying system. For this reason a serialized format (S-DISL) has been devised, which reduces size and complexity of an original DISL document without losing information. The number of tags is reduced through following means:

1. Merging of child elements into the parent tag
2. place Union of tags that require the same evaluation semantic
3. Discarding structural tags with no semantic meaning

The first method can be applied to elements where the number of children is finite and static. In that case, the child elements can be converted to a set of attributes for the parent element. The second method applies the same principle as the first one but is used to combine those parts that belong together with respect to the evaluation but have been distributed for better human readability. Some very few tags may be discarded at all as they convey no semantic meaning but just provide information for the human editor of the DISL document (e.g. the part element for logical structuring of the user interface). The tags for which none of the previous methods can be applied are kept intact. The following listing shows for example a part of the collected conditions in the SDISL format.

```
<cl cs="44">
  <c uid="91" cid="none" exp="equal" n="no" tA="variable" oA="58"
     tB="constant" oB="true"/>
  <c uid="92" cid="none" exp="equal" n="no" tA="variable" oA="58"
     tB="constant" oB="false"/>
  <c uid="93" cid="none" exp="equal" n="no" tA="variable" oA="59"
     tB="constant" oB="true"/>
  <c uid="94" cid="none" exp="equal" n="no" tA="variable" oA="59"
     tB="constant" oB="false"/>
  . . .
</cl>
```

The element <cl> stands for condition list and is basically a table of all the conditions that exist in the user interface description, while its attribute ”cs” reveals the number of the conditions in the list, so that the interpreter knows the size of the table it must set up. For each element of the UI specification a unique numerical identifier ”uid” is assigned instead of the manually selected ids in DISL in order to avoid conflicts when sourcing templates or when dealing with several (sub) interfaces at once. Looking at the first condition in the list, one can see that it checks for the equality of the variable with the id ”58” and the ”true”-constant.

While this encoding and re-sorting makes the processing of the specification easier, the limitations of elements and restrictions with the assignment of ids provide a reduction of the complete interface specification, so that it is more suitable for limited memory and slow networks. A further compression is gained after the assignment of

specific tokens for reserved words like ”variable” or ”constant” etc. The latter step is a similar process as proposed with binary XML (Martin and Jano 1999)(Martin and Jano, 1999). With these concepts applied, an SDISL renderer has been successfully implemented on a first-generation J2ME-enabled mobile phone (Mueller et al., 2004) and currently 32k Smart Cards are employed to carry complete additionally gzipped SDISL UI descriptions, embedded in a secure architecture (Schaefer et al., 2007).

7.4 IMPROVEMENTS TO UIML FOR VERSION 4.0

Having shown the goals and properties of the different available UIML tools, it is now time to review the most recent enhancements to the UIML specification, which are particularly useful for a human-centered software engineering process, namely variables, layout management and parameterizable templates.

7.4.1 Variables and Arithmetic for Improved Dialogs

As UIML is designed to represent all possible user interfaces and provides a clear separation of concerns, a core part consists of the definition of the behavior, which is analog to the dialog model in the sense of a control model in model-based user interface development. From the first version of UIML, the behavior part provided the possibility to specify event-based dialogs, such as reacting on a ”button-pressed” event. Furthermore the behavior could be used to define conditions which contain an event and some associated data. A prominent example is the selection of an item in a list box when an item-selected event fires and the value of the item can be extracted from a property. Only if the event hits, and the value is correct, such a condition evaluates to true and the action part can be executed. However events are transient by nature and properties are limited to their widgets they belong to. This can be a restriction for the UI developer in the sense that parts of the behavior which could belong to the UI have to be delegated to the application logic. Up to version 4.0, UIML had no means to capture and exploit information of the UI state, which allows for more powerful dialog models. For this purpose, variables have been introduced and together with simple arithmetic, the control model can be kept completely within the UIML document. As a most simple example for building a UI state machine, consider a dialog with only one button which toggles the state between on and off. The UIML code follows is illustrated in figure 7.9.

Let us focus on the behavior part. First, a Boolean variable OnOffState has been defined, which is eventually used to capture the state of this little dialog. The following variables are constants for true and false which are used later for comparisons and assignments. The actual behavior of the dialog is defined with two rules, one for an even number of button operations and one for an odd number. Both check whether a button has been pressed and then check if the current state matches one of the Boolean constants (e.g. for the even case, the OnOffState variable should equal to the EvenValue before the action can be executed. The action part then assigns to the OnOffVariable the inverse value in order to toggle the state. This example just employed comparison operations and assignments, but UIML 4.0 also supports some basic arithmetic which

```
<uiml>
<interface>
  <structure>
    <part id="button" class="G:Button"/>
  </structure>
  <style>
    <property part-name="button" name="g:text">ON/OFF</property>
  </style>
  <behavior>
    <variable id="OnOffState" type="boolean" reference="false">
      false
    </variable>
    <variable id="TrueValue" constant="true" type="boolean"
              reference="false">
      true
    </variable>
    <variable id="FalseValue" constant="true" type="boolean"
              reference="false">
      false
    </variable>
    <!-- If state == true and button pressed then state = false -->
    <rule id="buttonPushedEvent">
      <condition>
        <op nam="and">
          <event part-name="button" class="g:actionperformed">
          <op name="equals">
            <variable id="OnOffState"/>
            <variable id="TrueValue"/>
          </op>
        </op>
      </condition>
      <action>
        <variable id="OnOffState">
          <variable id="FalseValue"/>
        </variable>
      </action>
    </rule>
    <!-- If state == false and button pressed then state = true -->
    <rule id="buttonPushedOdd">
      <condition>
        <op nam="and">
          <event part-name="button" class="g:actionperformed">
          <op name="equals">
            <variable id="OnOffState"/>
            <variable id="FalseValue"/>
          </op>
        </op>
      </condition>
      <action>
        <variable id="OnOffState">
          <variable id="TrueValue"/>
        </variable>
      </action>
    </rule>
  </behavior>
</interface>
<peers>
  <presentation base="Generic\_1.2\_Harmonia\_1.0">
</peers>
</uiml>
```

Figure 7.9 UIML code of the simple UI state machine example.

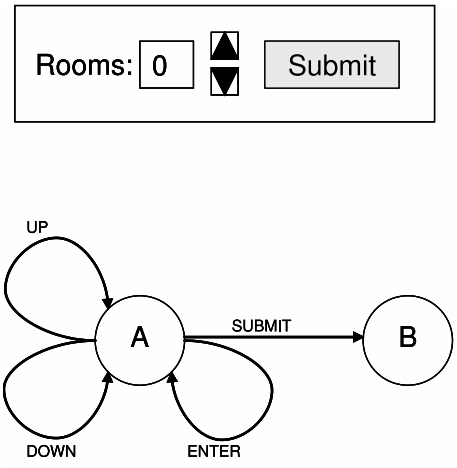

Figure 7.10 Room reservation form and corresponding state machine.

can be useful for several operations like counting the widget operations, comparison with timers, input validation etc. The latter use of variables is particularly useful in order to reduce communication with the backend logic, especially in a distributed case (e.g., client-server), and thus can add to the user acceptance, since systems can be designed to be more responsive already at a high level.

Consider for example a part of a hotel reservation form that checks the entered number of rooms with the policy of the hotel booking system. In this example, the hotel requires a minimum of one room and a maximum of four rooms to be booked by individuals.

Rooms can be entered directly in the text field or by using up- and down buttons, as illustrated in Figure 7.10. By pressing the buttons, the new room number is immediately checked and the value in the text field changed respectively. In case that the upper or lower bounds are reached, the value in the text field does not change. By pressing the ”submit” button, the value in the text field will be submitted to the backend. The advantage of this method is evident when the User Interface is connected via a network to the backend logic. By checking valid values on the client side no re-transmissions are required. This is equivalent to HTML-forms, where JavaScript would be used to check the input prior submission. The code fragment of Figure 7.11 shows how the input validation with the ”up”-button works in UIML 4.0.

The condition checks if the ”up”-button has been pressed and if the value of the variable curNoRooms is lower than the maximum allowed. Only if both is the case, the action-part can be processed in this case, simple arithmetic is used with the add operator which increments curNoRooms with the value of the variable step, which has been assigned to one beforehand. Additionally, the value in the text box is updated, as the property of editRooms is assigned the value of the variable curNoRooms.

Similarly to this rule, rules for checking the down button, direct text entry and submitting the data have to be established. Of course care has to be taken that this concept is

```
  <rule id="upButtonPressed>
        <condition>
          <op name="and">
            <event part-name="buttonUP" class="g:actionperformed">
            <op name="lessthan">
              <variable id="curNoRooms"/>
              <variable id="maxNoRooms"/>
            </op>
          </op>
        </condition>
        <action>
          <op name="add">
            <variable id="curNoRooms"/>
            <variable id="step"/>
          </op>
          <property part-name="editRooms" name="g:text">
            <variable id="curNoRooms"/>
          </property>
        </action>
</rule>
```

Figure 7.11 Part of the UIML code for the room reservation.

not misused to move parts of the application logic into the UI description, but considering that building program structures with XML is quite cumbersome, the risk is low that UI developers and application developers will produce conflicting overlaps.

7.4.2 Platform-independent Layouts

UIML achieves abstraction in many ways. The specification of user interface elements, interaction, content and the application logic are all platform-independent. However, the 3.0 version of the UIML specification has no support for flexible layout management, which results in platform-specific layout adjustments to be made by the designer for each of the target devices. UIML 4.0 supports a flexible layout management technique that ensures consistent layouts in a wide range of circumstances. The layout can still adjust to more extreme conditions without loosing consistency however.

If only the common characteristics of existing layout managers were to be extracted, the resulting layout mechanism would be so general it would only be suitable to create very simple layouts. Ideally, the general solution should support at least every layout that is possible by each of the layout managers that can be found for the individual specific widget sets. The generic layout extension we introduced for UIML 4.0 is based on the combination of spatial constraints and a constraint solving algorithm. Spatial constraints are sufficiently powerful to describe complex layouts (Badros et al., 2001) and allow us to offer several levels of abstractions. A spatial constraint is a mathematical relation concerning the location or size of different abstract interactors that needs to be maintained in the final user interface. The interface designer specifies the layout by defining constraints over the different user interface parts. This can be as

simple as stating *buttonA left-of labelB* to indicate buttonA should appear on the left of labelB. *Left-of* is an abstraction of a one dimensional mathematical relation that indicates the right side of buttonA on the horizontal axis should have a lower value than the left sife of labelB on the horizontal axis. Even simple constraint solvers can solve these kind of constraints.

A constraint solver can find a solution in a two- or three-dimensional solution space that adheres to these constraints. If only a two-dimensional solution space is supported, the interactors can be layout out on a two-dimensional canvas, but can not be put on top of each other (e.g. in ordered tab pages or partial overlaps). The UIML.net implementation includes Cassowary, an incremental constraint solver that efficiently solves systems of linear equalities and inequalities (Badros et al., 2001). The constraint solver is available in most of the programming languages that are used to implement UIML renderers and interpreters (Java, C++, C#, Python,...) and is distributed under a free software license. Future UIML implementations should have easy access to flexible constraint-based layout management.

Constraints are resolved on the level of the *abstract interaction objects*, so are *independent of the concrete representation* of the widgets. Constraints allow us to specify the layout in a declarative manner and integrate smoothly with UIML. The designer can focus only on *what* the desired layout exactly is, rather than *how* this layout is to be achieved. Furthermore, constraints allow partial specification of the layout, which can be combined with other partial specifications in a predictable way. For example, we can define that the container *selection* is *left-of* the container *content*. The selection and content containers can then each on their own specify the layout of their children. When a change in this layout requires the containers to grow, shrink or move, the upper-level layout constraints will be re-evaluated. This allows us to define generic layout patterns (Luyten et al., 2006). These define the layout of a number of containers, which can afterwards be filled in with a specific widget hierarchy using its own layout specification.

Figure 7.12 shows a music player interface rendered from a UIML document. This user interface can be used on different platforms and with different screen sizes as shown in figure 7.13. Without the layout management extension the designer could reuse a great deal of the UIML document for different platforms except the parts of the UIML document concerning the layout of the final user interface. The constraint-based layout technique solves this problem and makes it possible to reuse the interface design completely with different widget sets for different screen sizes without any manual intervention. If other behavior is required, a designer can still add or remove a constraint to obtain the envisioned effect, and is no longer bothered by any platform-specific problems while designing the interface.

Previously, designers had to specify a platform-specific layout for every instantiation of the user interface. This process relied on the careful and precise manual work of the designer in order to keep the different layouts consistent. Furthermore, this process introduced a lot of work, because for every new target platform, the layout had to be almost completely redesigned. The layout extension enables designers to create new layout templates and reuse them whenever appropriate. A layout template is nothing

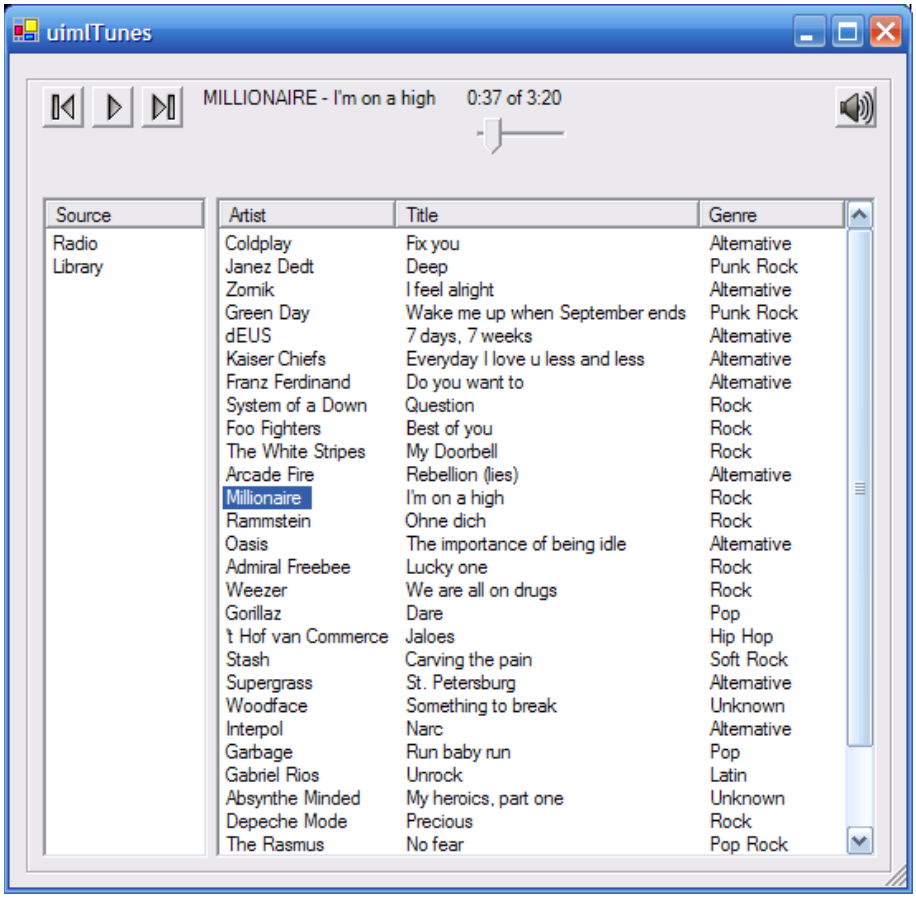

Figure 7.12 Fully functional music player interface rendered on desktop computer from UIML document

more than a set of layout constraints that can be applied to a set of parts from the structure section.

7.4.3 Template Parameterization

UIML provides a *template* mechanism that allows defining reusable components. A *reusable component* is a reusable part of the user interface: a combination of structure, style, behavior and/or content written down in UIML. They enable interface implementers to reuse part or all of their UI through the UIML <template> element. For example, many UIs for electronic commerce applications include a credit-card entry form. If such a form is described in UIML as a template, then it can be reused multiple times either within the same UI or across other UIs. This reduces the amount of

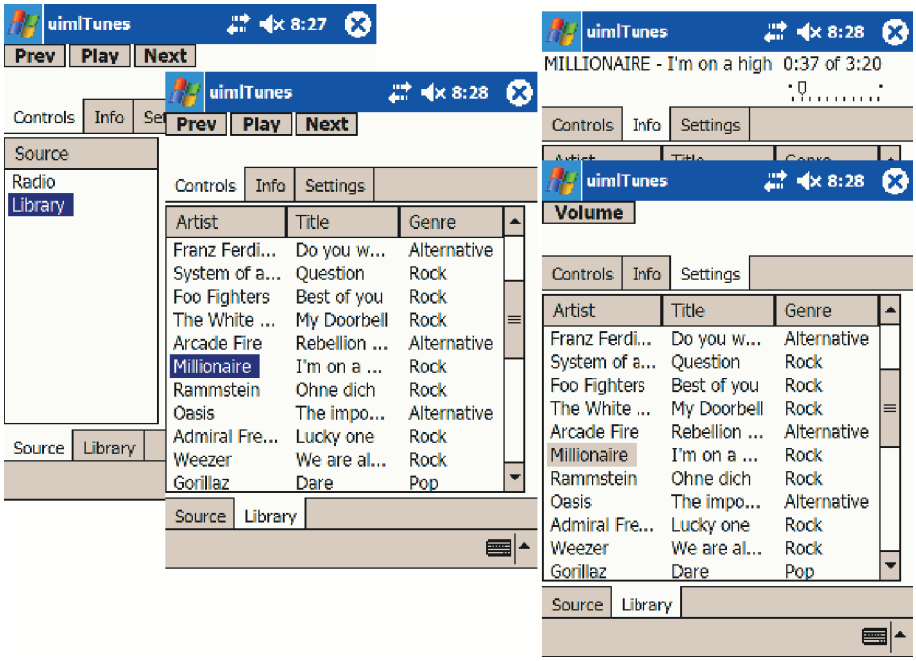

Figure 7.13 Fully functional music player interface rendered on PDA from UIML document

UIML code needed to develop a UI and also ensures a consistent presentation across enterprise-wide UIs.

The reusability of these components can be still be improved however. In this section we describe a method to enhance the reusability of the template mechanism by allowing more flexible relations between a template and its *sourcing document*; the part of the user interface that embeds the template.

We increased the expressive power of a template definition by allowing parameters to be passed from the sourcing document to the template. This allows a user interface designer to create a reusable user interface component that supports variations in structure, style, content and/or behavior. The components vary according to the parameters that are passed to it. In contrast with non-parameterized templates that provide immutable user interface components, a *parameterized template* allows us to describe a user interface pattern (van Welie et al., 2000). A pattern captures design knowledge for a specific problem that can be re-applied in other contexts, such as the credit-card entry form. Parameterized templates are very useful for describing reusable layout patterns, which we discussed in the previous section. These have the same purpose as the XUL layout patterns described by (Sinnig et al., 2004). Their approach uses the Velocity templating language to support variations in the patterns, while parameterized templates allow this to be built into UIML.

As a simple example, suppose we want to define the behavior of a calculator. When one of the buttons is pressed, the corresponding number is added to the display. The behavior rule for this would be:

```
  <rule>
  <condition>
    <event part-name="Button1" class="actionPerformed"/>
  </condition>
  <action>
    <property part-name="Display" name="text">
      <call name="utils.concatenate">
        <param>
          <property part-name="Display" name="text"/>
        </param>
        <param>
          <property part-name="Button1" name="label"/>
        </param>
      </call>
    </property>
  </action>
</rule>
```

We would need ten similar rules (one for each button), only varying in the part-name of the button. Although this scenario seems to be ideal for a reusable template, the 3.0 version of UIML is not capable of defining a template for these rules because they were slightly different.

Another aspect of the template mechanism where the reusability could be improved is the naming schemes that are used to avoid identifier duplicates when templates are merged into the sourcing documents part tree. Each element is associated with a *fully qualified identifier*, which is constructed by identifying the child's location in the UIML tree. The identifier is generated by starting with the <uiml>-element and tracing downward through the tree. The original element will be pre-pended with the *id* of every element above it in the UIML tree (e.g. "<interface id>__<structure id>__<grandparent id>__<parent id>__<original id>").

This creates a dependency between the sourcing document and the template, since they need to know each others structure to be able to refer to the correct elements. In the following example, a template will close the surrounding container (e.g. a window) when a certain button is clicked. It is clear that the template needs to know about the structure of the sourcing document, and can thus not be reused in other documents. The button's label (in this case "*Close*") is also hard-coded into the template, and cannot be easily changed.

```
  <part id="surrounding" class="Container">
  <part source="#closing" .../>
</part>
<template id="closing">
  <part>
    <part id="btn1" class="Button">
      <style>
        <property name="label">Close</property>
      </style>
```

```
      <behavior>
        <rule>
          <condition>
            <event class="actionPerformed"/>
          </condition>
          <action>
            <property part-name=
               "interf1_strct1_surrounding" name="visible">
              false
            </property>
          </action>
        </rule>
      </behavior>
    </part>
  </part>
</template>
```

In order to solve these issues, we introduced some concepts of traditional programming languages into the template specification. By passing parameters to a template it should be possible to establish stable, yet flexible relationships between a template and any UIML document.

The next example shows a parameterized template for the calculator behavior rules:

```
<template id="calculator_rule">
  <d-template-parameters>
    <d-template-param>button</d-template-param>
 </d-template-parameters>
  <rule>
    <condition>
      <event part-name="$button" class="actionPerformed"/>
    </condition>
    <action>
      <property part-name="Display" name="text">
        <call name="utils.concatenate">
          <param>
            <property part-name="Display" name="text"/>
          </param>
          <param>
            <property part-name="$button" property="label"/>
          </param>
        </call>
      </property>
    </action>
  </rule>
</template>
```

The template defines a parameter *button* , which will be used to fill in the correct reference in the label property and actionPerformed event. Note that if a parameter is used within a certain attribute, it is prefixed with a $-sign to distinguish it from normal references. To source this template, we pass the corresponding button as a parameter to it (in this case Button1):

```
<rule source="calculator_rule" how="replace">
  <template-parameters>
```

```
    <template-param id="button">Button1<template-param>
  </template-parameters>
</rule>
```

To parameterize the example where we want to close the surrounding container in the sourcing document, we introduce two parameters: one for the label of the button, and one for the surrounding container:

```
<template id="closing">
  <d-template-parameters>
    <d-template-param>label</d-template-param>
    <d-template-param>container</d-template-param>
  </d-template-parameters>
  <part>
    <part id="btn1" class="Button">
      <style>
        <property name="label">
          <template-param id="label"/>
</property>
      </style>
      <behavior>
        <rule>
          <condition>
            <event class="actionPerformed"/>
          </condition>
          <action>
            <property part-name="$container" name="visible">
              false
            </property>
          </action>
        </rule>
      </behavior>
    </part>
  </part>
```

The example shows that we can also use the parameters in any UIML tag (in this case, inside a property tag). Finally, to instantiate this template we provide values for the two parameters:

```
<part id="surrounding" class="Container">
  <part source="#closing" ...>
    <template-parameters>
      <template-param id="label">Close</template-param>
      <template-param id="container">
           surrounding
      </template-param>
    </template-parameters>
  </part>
</part>
```

In conclusion, *parameterized templates* improve the existing template mechanism in UIML to allow for full reusability since they are completely independent of the sourcing document.

7.5 UIML-RELATED STANDARDS

UIML is an answer to the question of what a declarative language would look like that could provide a canonical representation of any UI suitable for multi-platform, multi-lingual, and multi-modal UIs. This section describes the influences from W3C and other complimentary efforts on UIML, and comments on how UIML fits into these various technologies.

7.5.1 HTML, XML, CSS, WAI, and SOAP- Inspirations for UIML

Several W3C activities in 1997 – XML, HTML, CSS, and WAI – formed a catalyst of ideas that inspired the development of UIML. At that time a group of HCI developers in Blacksburg, Virginia who were frustrated with the difficulty of creating UIs in traditional imperative languages (e.g., C, C++) starting work on UIML using a number of insights from these W3C activities.

The success of HTML by 1997 in allowing non-programmers to design UIs with a rich user experience was a beacon of light to the team that designed the original UIML language: Could we start fresh, and design a new declarative language powerful enough to describe UIs that historically were built only in imperative programming languages and toolkits (e.g., C with X-windows, C++ with MFC, Java with AWT/Swing)? Doing so would bridge the gap between HTML, which allows easy design of UIs with limited interaction, and imperative languages, which allow design of rich UIs but only in the hands of experienced programmers.

In 1997, the first XML conference was held. XML is a meta-language, to which a vocabulary of element and attribute names must be added. XML could be standardized once, and was extensible because many vocabularies could be created by different groups of people. In designing UIML we realized that if a UI language was a meta-language, then it could potentially serve as a canonical representation of any UI. Hence UIML is a meta-language. By separately creating vocabularies for UIML, UIML could be devoid of bias toward UI metaphors, target devices (e.g., PCs, phones, PDAs), UI toolkits (e.g., Swing, MFC), and could be translated to various target languages (e.g., Java, HTML, VoiceXML).

The world was clearly on a trend to untether users from the desktop computer, allowing them to use a plethora of devices via growing wireless technologies. UIML recognized that a meta-language enables the creation of UI descriptions in a device-independent form.

Another influence by 1997 was Cascading Style Sheets, which could be viewed as the first step to created UI descriptions that are separated, or factored, into orthogonal components. The factoring was again a key to device-independent descriptions of UIs. The design of UIML started by asking what fundamentally are the orthogonal parts of a UI. The Model-View-Controller paradigm is a three-way separation. UIML arrived at a six-way separation (structure, style, content, behavior, APIs to components outside the UI, and mappings to UI toolkits) (Phanouriou, 2000).

The W3C's Web Accessibility Initiative, which also started in 1997, influenced UIML as well. The key to making documents and UIs accessible, according to WAI, is to

capture the author's intent. A language like HTML has ingrained into it a certain metaphor based on the printed page. What authors need is the ability to represent a UI using abstractions representing the semantic information they have, which cannot be rediscovered easily from markup like HTML. Again, a meta-language appeared to be a key element for UIML, because an author could define and work with his or her own abstractions in a vocabulary that the author creates.

A second influence of WAI was the recognition that scripting in HTML pages presents an obstacle to making documents portable across devices. The lesson learned for UIML's designers was that the behavior of a user's interaction with a UI should clearly be a separable component in a UI description.

The original work on SOAP in 1998 also influenced UIML. When SOAP was first proposed, it suggested that remote calls to objects could be done using XML. Therefore the actions in UIML's syntax for behavior description were designed to allow invocation of SOAP or other XML-based remote calls.

7.5.2 HCI - Another Influence on UIML

Aside from W3C, there was one other key influence on UIML: the field of Human-Computer Interaction (HCI). The design of UIs that work across devices requires a good design methodology. Much work has been done in the HCI field in UI design. There is also a body of literature called UI Management Systems, which include notations to represent UIs, and these heavily influenced the design of UIML (especially the question of how to represent user interaction with a UI in a canonical form).

Our expectation is that work on design techniques for UIs will produce a number of tools and UI design languages. UIML was not necessarily intended as a UI design language (although it can be used as such), but rather as a language for UI implementation. Therefore UI design tools could represent a design in a design language, and then transform a UI in a design language to a canonical representation for UI implementation, namely UIML. If Integrated Development Environments (IDEs) and Web page design tools could read UIML, then the world would have a complete path for computer-assisted design and implementation of multi-platform UIs.

7.5.3 How UIML Fits W3C Architecture Today

Dave Raggett in his talk at the W3C Workshop on Web Device Independence (Bristol, Oct. 2000) proposed that there was a need for a layer that can adapt a UI to the particular XML language used by a target device. UIML is an element of this device adaptation layer, but not a complete solution. For example, there may be transform algorithms that transform the interface description (e.g., in UIML) to take into account device characteristics.

Without a single canonical language to represent UIs at this layer (regardless of whether it is UIML), then one must create transforms for multiple languages. Obviously if it is possible to have one language at this layer, the construction of reusable transforms is simplified.

One way to apply UIML at this layer is to use multiple vocabularies with UIML, and transform from UIML using one vocabulary to UIML using another vocabulary. For example, one may start with a UI description using a generic vocabulary (e.g., a vocabulary whose abstractions can be mapped to a variety of devices). Perhaps the UI was authored with this generic vocabulary to facilitate accessibility. A transform algorithm, guided by a rule base that takes into account characteristics of different devices, can then be used to map UIML with the generic vocabulary to UIML with a vocabulary specific to a particular device. This technique has been implemented to adapt UIs to various versions of Web browsers (e.g., to give a similar appearance to UIs for HTML 3.2 vs. HTML 4.0 browsers). The UIML produced by the Device Adaptation layer can then be rendered to a particular XML language (e.g., by a rendering program that compiles UIML into XHTML, or UIML into VoiceXML).

7.5.4 The Path Towards Separation in User Interfaces

The evolution of W3C specifications in the UI area has followed a path of gradually separating a UI description into orthogonal parts:

Up until HTML 3.2, there was no separation.

In HTML4, the style was separated (via CSS and XSL-FO).

In XForms, the portion of a document that represents a form was separated.

In XML Events, events were separated.

As stated earlier, UIML separates a UI into six parts, answering these six questions:

1. What are the parts that constitute the structure of the UI?
2. What is the presentation style of the parts?
3. What is the content associated with the parts?
4. What is the behavior of the UI when a user interacts with the UI?
5. What is the API of components outside the UI with which the UI interacts?
6. What is the mapping of the vocabulary to a target UI toolkit or markup language?

These six questions are answered in UIML's structure, style, content, behavior, logic, and presentation elements, respectively.

Therefore this fundamental design decision in UIML is compatible with the path being followed by W3C. UIML should provide W3C working groups with an example of what will ultimately be reached as this path toward separation is followed in the future.

7.6 CONCLUSION

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 to focus on defining the mappings and transforms that enable UIML to be converted into the appropriate deployment language. We have found that this approach improves the software engineering aspects of UI engineering by improving reusability through

modular templates, enabling rapid prototyping by empowering the usability engineer to produce developer level code through automated tools, and separating both concerns within the UI design and platform idiosyncrasies from the abstract UI design.

Our experience indicates that adding layers of abstraction in the software engineering process can be especially beneficial when the systems and interfaces are complex. In these cases, 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 UIML 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 representation. The primary goal of UIML's designers has been to create such a meta-language as an open standard for UI definition. OASIS established a UIML Technical Committee (TC) that has examined UIML and is in the process of standardizing the language.

Acknowledgements

A. Coyette and J. Vanderdonckt acknowledge the support of the SIMILAR European network of excellence (`www.similar.cc`) on multimodal interfaces funded by European Commission. Part of the research at EDM is funded by EFRO (European Fund for Regional Development), the Flemish Government and the Flemish Interdisciplinary institute for Broadband Technology (IBBT). Work on the most relevant products in the LiquidApps^{TM} suite is supported by NAVAIR contracts N00421-04-C-0030, N68335-06-C-0010, and N68335-05-C-0029; NAVSEA contract N00164-06-C-6093; Office of Naval Research contract N00014-06-M-0047; and Missile Defense Agency contract W9113M-06-C-0041.

References

Abrams, M., Phanouriou, C., Batongbacal, A. L., Williams, S. M., and Shuster, J. E. (1999). UIML: An appliance-independent XML user interface language. *Computer Networks*, 31(11-16):1695–1708.

Ali, M. F., Pérez-Quiñones, M. A., Abrams, M., and Shell, E. (2002). Building multi-platform user interfaces with UIML. In Kolski, C. and Vanderdonckt, J., editors, *CADUI Computer-Aided Design of User Interfaces III, Proceedings of the Fourth International Conference on Computer-Aided Design of User Interfaces, May, 15-17, 2002, Valenciennes, France*, pages 255–266. Kluwer.

Coyette, A. and Vanderdonckt, J. (2005). A sketching tool for designing anyuser, anyplatform, anywhere user interfaces. In Costabile, M. F. and Paternò, F., editors, *INTERACT*, volume 3585 of *Lecture Notes in Computer Science*, pages 550–564. Springer.

Eisenstein, J., Vanderdonckt, J., and Puerta, A. (2001). Applying model-based techniques to the development of UIs for mobile computers. In *Proceedings of the 2001 International Conference on Intelligent User Interfaces*, pages 69–76, New York. ACM Press.

Hartson, H. R. and Hix, D. (1989). Toward empirically derived methodologies and tools for human-computer interface development. *International Journal of Man-Machine Studies*, 31(4):477–494.

Limbourg, Q. and Vanderdonckt, J. (2004). UsiXML: A user interface description language supporting mul-tiple levels of independence. In Matera, M. and Comai, S., editors, *Engineering Advanced Web Applications*, pages 325–338. Rinton Press, Paramus.

Luyten, K. and Coninx, K. (2004). UIML.NET: an open UIML renderer for the.net framework. In Jacob, R. J. K., Limbourg, Q., and Vanderdonckt, J., editors, *CADUI*, pages 257–268. Kluwer.

Martin, B. and Jano, B. (1999). Wap binary xml content format. World Wide Web Consortium. W3C NOTE.

Mueller, W., Schaefer, R., and Bleul, S. (2004). Interactive multimodal user interfaces for mobile devices. In *HICSS*.

Phanouriou, C. (2000). *UIML: A Device-Independent User Interface Markup Language*. PhD thesis, Vermount University. Available at `http://scholar.lib.vt.edu/theses/available/etd-08122000-19510051/unrestricted/PhanouriouETD.pdf`.

Puerta, A. R. and Eisenstein, J. (2002). XIML: a common representation for interaction data. In *IUI*, pages 216–217.

Schaefer, R., Mueller, W., López, M., and Sánchez, D. (2007). Device independent user interfaces for smart cards. Technical report, C-LAB Report.

Vanderdonckt, J. and Bodart, F. (1993). Encapsulating knowledge for intelligent automatic interaction objects selection. In *Proceedings of ACM INTERCHI'93 Conference on Human Factors in Computing Systems, Amsterdam*, pages 424–429. ACM Press, New York.

Zimmermann, G., Vanderheiden, G. C., and Gilman, A. S. (2002). Universal remote console - prototyping for the alternate interface access standard. In Carbonell, N. and Stephanidis, C., editors, *User Interfaces for All*, volume 2615 of *Lecture Notes in Computer Science*, pages 524–531, Paris, France. Springer.