A MDA-Compliant Environment for Developing User Interfaces of Information Systems

Jean Vanderdonckt

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

Abstract. To cope with the ever increasing diversity of markup languages, programming languages, tool kits and interface development environments, conceptual modeling of user interfaces could bring a framework for specifying, designing, and developing user interfaces at a level of abstraction that is higher than the level where code is merely manipulated. For this purpose, a complete environment is presented based on conceptual modeling of user interfaces of information systems structured around three axes: the models that characterize a user interface from the end user's viewpoint and the specification language that allows designers to specify such interfaces, the method for developing interfaces in forward, reverse, and lateral engineering based on these models, and a suite of tools that support designers in applying the method based on the models. This environment is compatible with the Model-Driven Architecture recommendations in the sense that all models adhere to the principle of separation of concerns and are based on model transformation between the MDA levels. The models and the transformations of these models are all expressed in UsiXML (User Interface eXtensible Markup Language) and maintained in a model repository that can be accessed by the suite of tools. Thanks to this environment, it is possible to quickly develop and deploy a wide array of user interfaces for different computing platforms, for different interaction modalities, for different markup and programming languages, and for various contexts of use.

1 Introduction

Today, the development of the User Interface (UI) of interactive applications, whether they are an information system or a complex, possibly safety-critical system, poses a series of unprecedented challenges due to the multiplication of several variables [10]:

- *Diversity of users*: the end users of a same interactive application could no longer be considered similar as they exhibit various skills, capabilities, levels of experience and preferences that should be reflected in the UI. Instead of having one single UI for all users, a family of different UIs should be developed to cope with the differences of multiple categories of users, including those who are impaired.
- *Richness of cultures*: when an interactive application is going to be global, its UI cannot remain the same for all languages, countries, and cultures. Rather, it can be

submitted to a process of localization to tailor the UI to particular constraints or to a process of globalization to adapt the UI to the largest population possible.

- *Complexity of interaction devices and styles*: Human-Computer Interaction (HCI) is an area known for dealing with a wide variety of interaction devices (e.g., bi-manual mouse, 3D pointers, laser pointer, phantom) and styles (3D tracking, eye tracking, gesture recognition, speech recognition and synthesis). The handling of events generated by these devices and their sound incorporation into an interaction style requires many programming skills that often go beyond the classical capabilities of an average developer of an information system. Even more, when several modalities are combines, as in a multimodal application, this complexity is decupled. Same for virtual reality, augmented reality and mixed reality applications.
- *Heterogeneousness of computing platforms*: the market of computing platforms is submitted to a constant introduction of new computing platforms, each one coming with a new set of constraints to be imposed on the UI that should run on it. For instance, a significant constraint is the screen resolution and the interaction capabilities that largely vary depending on the computing platform: mobile phone, smartphone, Pocket PC, Blackberry, Handbag PC, Tablet PC, interactive kiosk, laptop, desktop, multi-displays workstation, projected UI, wall screen. All these computing platforms do not necessarily run the same operating system and the UI is not necessarily developed with the same markup language (e.g., WML, cHTML, HTML, DHTML, VoiceXML, X+V, VRML97, X3D) or programming language (e.g., Visual Basic, C++, Java, C#). Even when a same language could be used, several peculiarities are present on each platform, thus preventing the developer from reusing code from one platform to another. The typical end user is using today at least three platforms, sometimes with some synchronization between.
- *Multiplicity of working environments*: end users are nowadays confronted to a series of different physical environments where they are supposed to work with the same reliability and efficiency. But when the environment becomes more constraining, e.g., in stress, in noise, in light, in availability of network resources, the UI is not necessarily adapted to these variations.
- *Multiplicity of contexts of use*: if a given context of use is defined as a particular user category working with a given platform in a specific environment, then the array of potential contexts of use explodes. Of course, not all the contextual variations should be considered and interpreted in a significant change of the UI, but at least a reasonable amount of differences exist for context-aware applications. From a user's perspective, various scenarios may occur [1,3]:

    1. Users may move between different computing platforms whilst involved in a task: when buying a movie on DVD a user might initially search for it from her desktop computer, read the reviews of the DVD on a PDA on the train on the way home from work, and then order it using a WAP-enabled mobile phone.
    2. The context of use may change whilst the user is interacting: the train may go into a dark tunnel so the screen of the PDA dims, the noise level will rise so volume of audio feedback increases so it can still be heard.

3. Users may want to collaborate on a task using heterogeneous computing platforms: the user decides to phone up a friend who has seen the movie and look at the reviews with her, one person using WebTV and the other using a laptop, so the same information is presented radically differently.

- *Multiplicity of software architectures*: due to the above variations, the UI should be developed with dedicated software architecture in mind (e.g., [5]) that is explicitly addressing the variations considered as for mobile computing, ubiquitous computing, context-aware applications (e.g., [1]).

Therefore, it is rather difficult to obtain a UI that addresses these variations while avoiding reproducing multiple UIs for different contexts of use without factoring out the common parts. In addition, when in the past, it was possible to code a UI by hand, today this empirical, opportunistic approach is no longer viable. All these reasons and others stem for a methodology for *User Interface Engineering*. This discipline is located midway between Software Engineering (SE), Human-Computer Interaction (HCI) and Human Factors (HF). Its primary goal is to develop a methodology for developing the UI throughout the development life cycle that can be articulated with traditional SE concepts. This methodology consists of:

1. A series of models pertaining to various facets of the UI such as: task, domain, user, presentation, dialog, platform, context of use, etc. These models will be defined in Section 2 and located on a reference framework. These models are uniformly and univocally expressed according to a single Specification Language, described in Section 3.
2. A step-wise method towards Computer-Aided Design of User Interfaces (CADUI) based on any combination of the above models. Section 4 will define this method by combination of models and model transformations so as to be compliant with the Model-Driven Architecture, to support Model-Driven Development (MDD).
3. A suite of software engineering tools that supports the designer and the developer during the development life cycle according to the method. A subset of these tools will be illustrated in Section 5.

Section 6 will summarize the main benefits of the MDA-compliant environment.

2 Models

Our methodology is explicitly based on the Cameleon Reference Framework [3], which defines UI development steps for multi-context interactive applications. Its simplified version, reproduced in Fig. 1, structures development processes for two contexts of use into four development steps (each development step being able to manipulate any specific artifact of interest as a model or a UI representation):

1. *Final UI* (FUI): is the operational UI i.e. any UI running on a particular computing platform either by interpretation (e.g., through a Web browser) or by execution (e.g., after compilation of code in an interactive development environment).

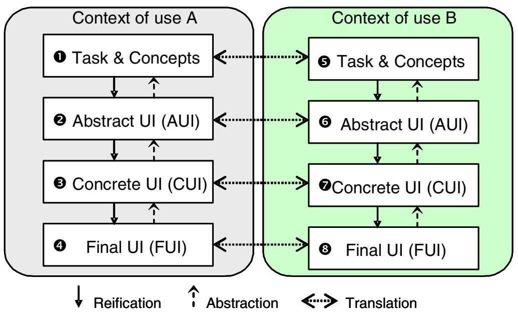
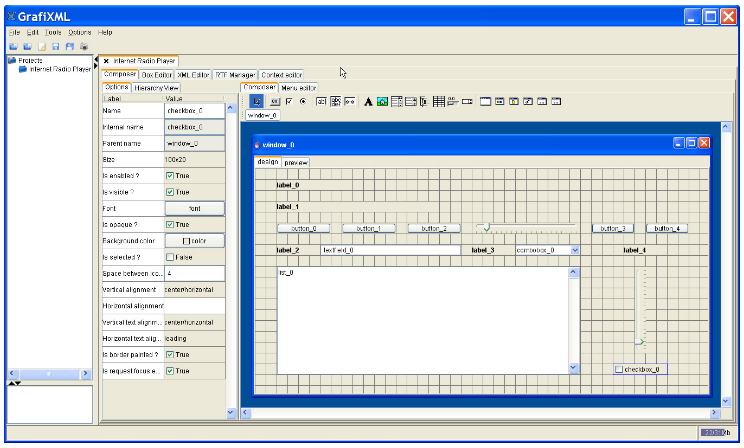


Fig. 1. The Simplified User Interface Reference Framework [3]

2. *Concrete UI* (CUI): concretizes an abstract UI for a given context of use into Concrete Interaction Objects (CIOs) [19] so as to define widgets layout and interface navigation. It abstracts a FUI into a UI definition that is independent of any computing platform. Although a CUI makes explicit the final Look & Feel of a FUI, it is still a mock-up that runs only within a particular environment (Fig. 2). A CUI can also be considered as a reification of an AUI at the upper level and an abstraction of the FUI with respect to the platform. For example, in Envir3D [18], the CUI consists of a description of traditional 2D widgets with mappings to 3D by relying on different mechanisms when such a mapping is possible.
3. *Abstract UI* (AUI): defines abstract containers and individual components [12,13], two forms of Abstract Interaction Objects [19] by grouping subtasks according to various criteria (e.g., task model structural patterns, cognitive load analysis, semantic relationships identification), a navigation scheme between the container and selects abstract individual component for each concept so that they are independent of any modality. An AUI abstracts 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. An AUI is considered as an abstraction of a CUI with respect to interaction modality. At this level, the UI mainly consists of input/output definitions, along with actions that need to be performed on this information (Fig. 3). The AUI is mainly based on the Canonical Abstract Prototypes [4].
4. *Task & Concepts* (T&C): describe the various user's tasks to be carried out and the domain-oriented concepts as they are required by these tasks to be performed. These objects are considered as instances of classes representing the concepts. Fig. 4 represents a potential task model representing the end user's viewpoint for an Internet Radio Player, based on LOTOS operators.

This framework also exhibits three types of transformation types: (1,2) *Abstraction* (respectively, *Reification*) is a process of eliciting artifacts that are more abstrct (respectively, concrete) than the artifacts that serve as input to this process. Abstraction


GrafiXML
File Edit Tools Options Help
Projects
Internet Radio Player
Composer Box Editor XML Editor RTF Manager Context editor
Options Hierarchy View
Composer Menu editor
window_0
design preview
label_0
label_1
button_0 button_1 button_2 button_3 button_4
label_2 textfield_0 label_3 combobox_0 label_4
list_0
checkbox_0

Label	Value
Name	checkbox_0
Internal name	checkbox_0
Parent name	window_0
Size	100x20
Is enabled ?	✓ True
Is visible ?	✓ True
Font	font
Is opaque ?	✓ True
Background color	☐ color
Is selected ?	☐ False
Space between ico...	4
Vertical alignment	center/horizontal
Horizontal alignment	
Vertical text alignm...	center/horizontal
Horizontal text alig...	leading
Is border painted ?	✓ True
Is request focus e...	✓ True

Fig. 2. A concrete UI of an Internet Radio Player [15] in GrafiXML

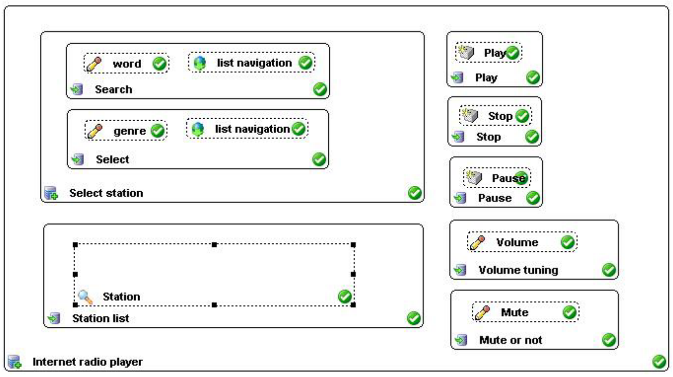


Fig. 3. A Abstract User Interface of an Internet Radio Player [15] in IdealXML [16]

is the opposite of reification. (3) *Translation* is a process that elicits artifacts intended for a particular context of use from artifacts of a similar development step but aimed at a different context of use. Therefore, when there is a need to switch from one context of use (e.g., one platform) to another one (e.g., another platform), the development process can cope with adaptation to the new context of use at any level of abstraction. Of course, the higher the level of abstraction the adaptation is performed, the more flexibility we have in the resulting processes.

With respect to this framework, multi-path UI development [12,13] refers to a UI engineering method and tool that enables a designer to (1) start a development activity from any entry point of the reference framework (Fig. 1), (2) get substantial support in the performance of transformation types and their combinations as found in Fig. 1. Several interesting development paths can be expressed on this framework since not

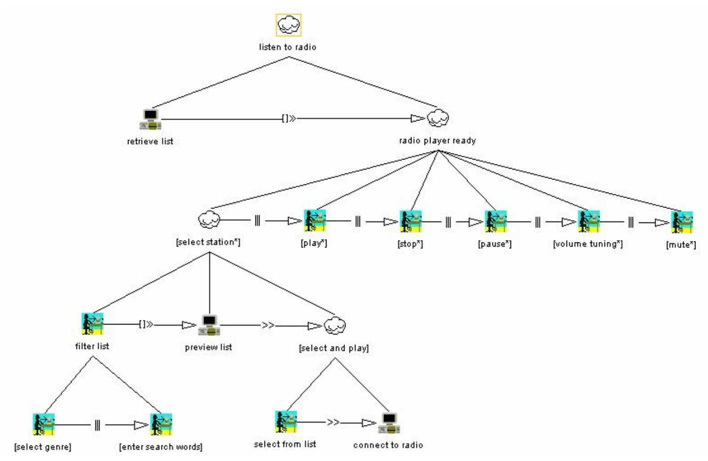


Fig. 4. A task model of an Internet Radio Player [15] in IdealXML [16]

all steps should be achieved in a sequential ordering dictated by the levels. Instead, locating *what* steps are performed, when, from which entry point and toward what subsequent step are important. According to Fig. 1, *transcoding* tools start with a FUI for a source context of use (❹) and transforms it into another FUI for a target context (❽). Similarly, *portability* tools start with a CUI for a source context (❸) and transforms it into another CUI for another context (❼), that in turn leads to a new FUI for that context (❽). To overcome shortcomings identified for these tools, there is a need to raise the level of abstraction by working at the AUI level. *UI Reverse Engineering* abstracts any initial FUI (❹) into concepts and relationships denoting a AUI (❷), which can then be translated into a new AUI (❻) by taking into account constraints and opportunities for the new context. *UI Forward Engineering* then exploits this AUI (❻) to regenerate a new AUI adapted to this platform, by recomposing the CUI (❼) which in turn is reified in an executable FUI (❽). In general, UI reverse engineering is any combination of abstraction relationships starting from a FUI (❹), a CUI (❸) or an AUI (❷). UI forward engineering is any combination of reification relationships starting from T&C, AUI or CUI. Similarly, *UI Lateral Engineering* is responsible for applying any translation at any level of abstraction to transform the artifacts existing at the level where we are for another context of use at the same level. For instance, when a designer has already designed a UI for, let us say a desktop, and wants to design a corresponding UI for a Pocket PC, she may need to apply *Graceful Degradation* techniques [7], which consist of a series of transformations to support the translation from the desktop to the PocketPC, while taking into account the constraints imposed by the new platform.

So far, to support conceptual modeling of UIs and to describe UIs at various levels of abstractions, the following models have been involved (Fig. 5) [13]:

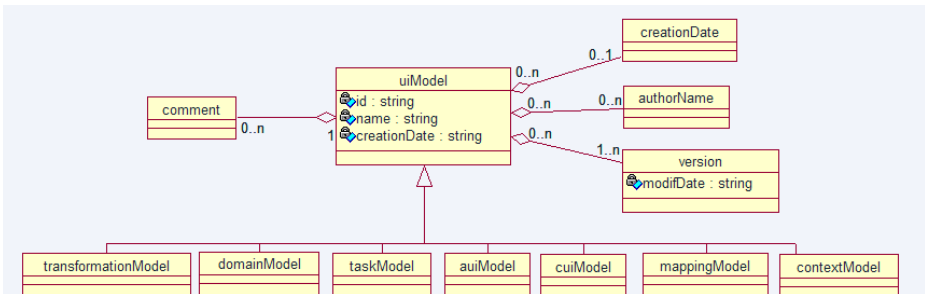


Fig. 5. The collection of models for specifying a user interface [12,13]

- **taskModel**: is a model describing the interactive task as viewed by the end user interacting with the system. A task model represents a decomposition of tasks into sub-tasks linked with task relationships. Therefore, the decomposition relationship is the privileged relationship to express this hierarchy, while temporal relationships express the temporal constraints between sub-tasks of a same parent task.
- **domainModel**: is a description of the classes of objects manipulated by a user while interacting with a system. Typically, it could be a UML class diagram, an entity-relationship-attribute model, or an object-oriented model.
- **mappingModel**: is a model containing a series of related mappings between models or elements of models. A mapping model serves to gather a set of inter-model relationships that are semantically related. It expresses reification, abstraction, and translation. In addition, other mappings [13] are defined and could be defined.
- **contextModel**: is a model describing the three aspects of a context of use in which a end user is carrying out an interactive task with a specific computing platform in a given surrounding environment [3]. Consequently, a context model consists of a *user model*, a *platform model* [5], and an *environment model*.
- **auiModel**: is the model describing the UI at the abstract level as previously defined.
- **cuiModel**: is the model describing the UI at the concrete level as previously defined.
- **uiModel**: is the topmost superclass containing common features shared by all component models of a UI. A uiModel may consist of a list of component model sin any order and any number, such as task model, a domain model, an abstract UI model, a concrete UI model, mapping model, and context model. A user interface model needs not include one of each model component. Moreover, there may be more than one of a particular kind of model component.

The conceptual modeling activities that reached to the meta-model of the cuiModel represented a significant amount of work and is therefore detailed further in the next subsection. The transformation model is the only remaining model: as such, it is defined in the second subsection.

2.1 The cuiModel

A CUI is assumed to be described without any reference to any particular computing platform or toolkit of that platform. For this purpose, a CUI model consists of a hierarchical decomposition of CIOs. A *Concrete Interaction Object* (CIO) is defined as any UI entity that users can perceive such as text, image, animation and/or manipulate such as a push button, a list box, or a check box [12,13,19]. A CIO is characterized by various attributes such as, but not limited to: *id*, *name*, *icon*, *content*, *defaultContent*, *defaultValue*.

Since a CIO is independent of any computing platform, we do not know yet which interaction modality is used on that platform. Therefore, each CIO can be sub-typed into sub-CIOs depending on the interaction modality chosen: *graphicalCIO* for GUIs, *auditoryCIO* for vocal interfaces, *3DCIO* for 3D UIs, etc. In this paper, we focus on graphical CIO since they form the basic elements of a traditional 2D GUI or a 3D, virtual UI. Each *graphicalCIO* inherits from the above CIO properties and has specific attributes such as: *isVisible*, *isEnabled*, *fgColor* and *bgColor* to depict foreground and background colors, etc.

Each *graphicalCIO* is then sub-typed into one of the two possible categories (Fig. 6): *graphicalContainer* for all widgets containing other widgets such as page, window, frame, dialog box, table, box and their related decomposition or *graphicalIndividualComponent* for all other traditional widgets that are typically found in such containers. A graphicalIndividualComponent cannot be further decomposed. The model supports a series of widgets defined as graphicalIndividualComponents such as: *textComponent*, *videoComponent*, *imageComponent*, *imageZone*, *radioButton*, *toggleButton*, *icon*, *checkbox*, *item*, *comboBox*, *button*, *tree*, *menu*, *menuItem*, *drawingCanvas*, *colorPicker*, *hourPicker*, *datePicker*, *filePicker*, *progressionBar*, *slider*, and *cursor*.

Thanks to this progressive inheritance mechanism, every final elements of the CUI inherits from the upper properties depending on the category they belong to. The properties that have been chosen to populate the CUI level have been decided because they belong to the intersection of property sets of major toolkits and window managers, such as Windows GDI, Java AWT and Swing, HTML. Of course, only properties of high common interest were kept. In this way, a CIO can be specified independently from the fact that it will be further rendered in HTML, VRML or Java. This quality is often referred to as the property of *platform independence*.

Similar abstractions exist for auditory, vocal, 3D, virtual, and augmented reality interfaces.

2.2 The TransformationModel

Graph Transformation (GT) techniques were chosen to formalize explicit transformations between any pair of models [13] (except from the FUI level), because it is (1) **Visual**: every element within a GT based language has a graphical syntax; (2) **Formal**: GT is based on a sound mathematical formalism (algebraic definition of graphs;

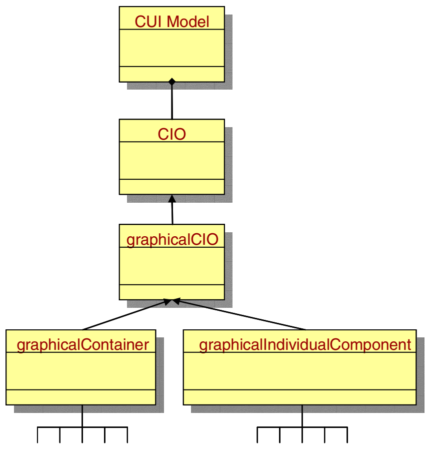


Fig. 6. Decomposition of a CUI model into concepts

and category theory) and enables verifying formal properties on represented artefacts; (3) **Seamless**: it allows representing manipulated artefacts and rules within a single formalism. Furthermore, the formalism applies equally to all levels of abstraction of uiModel (Fig. 1). The model collection (Fig. 5) is structured according to the four basic levels of abstractions defined in the Cameleon Reference Framework [3] that is intended to express the UI development life cycle for context-sensitive interactive applications. The FUI level is the only model that cannot be supported by graph transformations because it would have supposed that any markup or programming language to be supported should have been expressed in a meta-model to support transformations between meta-models: the one of the initial language to the one of our specification language (Section 3). It was observed that to address this problem, the powerfulness of GT techniques was not needed and surpassed by far other experienced techniques, such a derivation rules [2].

3 Specification Language

In order to specify the different UI aspects and related models, a specification language is needed that allow designers and developers to exchange, communicate, and share fragments of specifications and that enables tools to operate on these specifications. Therefore, a User Interface Description Language (UIDL) is required that satisfies at least the following important requirements: *software processability* (the UIDL should be precise enough to enable computational processing of the specifications by an automaton, in particular to interpret or execute them), *expressiveness* (the UIDL should be expressive enough to support common SE techniques such as model derivation, model transformation, mapping), *standard format* (the UIDL should be expressed in a format that maximizes its exchange among stakeholders), *human readability* (the UIDL should be as much as possible legible and understandable by a human agent), *concision* (the UIDL should be compact enough to be easily exchanged among interested parties). Since software processability and expressiveness are the two most important requirements, trade-offs could be accepted to satisfy those two requirements, while decreasing the value of the two other requirements.

To address the above requirements, we introduced UsiXML (which stands for USer Interface eXtensible Markup Language – http://www.usixml.org), a XML-compliant markup language that describes the UI for multiple contexts of use such as Character User Interfaces (CUIs), Graphical User Interfaces (GUIs), Auditory and Vocal User Interfaces, Virtual Reality, and Multimodal User Interfaces:

- UsiXML is primarily intended for non-developers, such as analysts, specifiers, designers, human factors experts, project leaders, and novice programmers.
- UsiXML can be equally used by experienced designers and developers.
- Thanks to UsiXML, non-developers can shape the UI of any new interactive application by specifying, describing it in the UIDL, without requiring programming skills usually found in markup languages and programming languages.
- UsiXML consists of a declarative UIDL that capturing the essence of what a UI is or should be independently of physical characteristics.
- UsiXML describes at a high level of abstraction the constituting UI elements of an application: widgets, controls, containers, modalities, and interaction techniques.
- UsiXML allows cross-toolkit development of an interactive application.
- A UI of any UsiXML-compliant application runs in all toolkits that implement it: compilers and interpreters.
- UsiXML supports *device independence*: a UI can be described in a way that remains autonomous with respect to the devices used in the interactions (e.g., mouse, screen, keyboard, voice recognition system). In case of need, a reference to a particular device can be incorporated.
- UsiXML supports *platform independence*: a UI can be described in a way that remains autonomous with respect to the various existing computing platforms (e.g., mobile phone, Pocket PC, Tablet PC, kiosk, laptop, desktop, and wall screen). In case of need, a reference to a particular computing platform can be incorporated.
- UsiXML supports *modality independence*: a UI can be described in a way that remains independent of any interaction modality (e.g., graphical interaction, vocal interaction, 3D interaction, virtual reality interaction). In case of need, a reference to a particular modality can be incorporated.
- UsiXML allows reusing elements previously described in anterior UIs to compose a UI in new applications.

On the other hand, it is not supposed to cover all features of all types of UI:

- UsiXML does not want to introduce yet another language for UI implementation. Instead, it proposes the integration of some of these formats: cHTML, WML, HTML, XHTML, VoiceXML, VRML, Java, C++,.... It is up to the underlying implementation to support the transformation of UsiXML into such a format.
- UsiXML does not describe the low-level details of elements involved in the various modalities, such as operating system attributes, events, and primitives.

- UsiXML cannot be rendered nor executed by its own: it relies on an implementation in any third-party rendering engine.
- UsiXML does not want to support all attributes, events, and primitives of all widgets existing in nearly all toolkits. Instead, it is intended to support a common subset of them that is believed to be representative and significant.

For the moment, the semantics of UsiXML are defined according to the above models and relationships in terms of a UML class diagram, representing the meta-model of the UIDL. All class diagrams are maintained in Rational Rose and lead to the definition of ML schemas that are available at http://www.usixml.org/index.php?view=page &idpage=5 thanks to a series of systematic transformations. Therefore, any UI specification is expressed in UsiXML that is in turn compliant with the XML schemas.

4 Method

So far, many attempts to establish a comprehensive model-based approach for developing the UI have been launched [10,11,20,21], but only a few of them is MDA-compliant: form information related task (what are the actions carried out by the user), domain (what are the objects manipulated in this task), user (who is the user), platform (what is the computing platform), environment (in which environment is the user working), the presentation, the dialog, the help, the tutorial of one or many UIs should be derived. Today, no consensus has been reached and no method has really emerged from these initiatives, namely by lack of standardization. Since 1997, the Object Management Group (OMG – www.omg.org) has launched an initiative called Model Driven Architecture (MDA) to support the development of complex, large, interactive software systems providing a standardized architecture with which:

- Systems can easily evolve to address constantly evolving user requirements.
- Old, current and new technologies can be harmonized.
- Business logic can be maintained constant or evolving independently of the technological changes.
- Legacy systems can be integrated and unified with new systems.

In this approach, models are applied in all steps of development up to a target platform, providing source code, deployment and configuration files,… MDA has been applied to many kinds of business problems and integrated with a wide array of other common computing technologies, including the area of UIs.

In MDA, a systematic method is recommended to drive the development life cycle to guarantee some form of quality of the resulting software system. Four principles underlie the OMG's view of MDA [14]:

1. Models are expressed in a well-formed unified notation and form the cornerstone to understanding software systems for enterprise scale information systems. The semantics of the models are based on meta-models.
2. The building of software systems can be organized around a set of models by applying a series of transformations between models, organized into an architectural framework of layers and transformations: model-to-model transformations

support any change between models while model-to-code transformation are typically associated with code production, automated or not.
3. A formal underpinning for describing models in a set of meta-models facilitates meaningful integration and transformation among models, and is the basis for automation through software.
4. Acceptance and adoption of this model-driven approach requires industry standards to provide openness to consumers, and foster competition among vendors.

Our UI engineering methodology, based on UsiXML (Section 3), is compliant with these four principles in the following way:

1. All involved models are expressed in UsiXML, a well-formed UIDL based on XML schema. The semantics of the UsiXML models are based on meta-models expressed in terms of UML class diagrams, from which the XML schema definition are derived. Right now, there is no automation between the initial definition of the semantics and their derivation into XML schemas. Only a systematic method is used for each new release.
2. All model-to-model transformations are themselves specified in UsiXML to keep only one UIDL throughout the development life cycle. Model-to-code transformation is ensured by appropriate tools (see Section 5) that produce code for the target context of use or platform. For reverse engineering, code-to-model transformations are mainly achieved by *derivation rules* that are based on the mapping between the meta-model of the source language (e.g., HTML, WML, VoiceXML) and the meta-model of the target language (i.e., here UsiXML). So far, derivation rules have been proved powerful enough to express reverse engineering rules for UI, while keeping a relative concision.
3. All transformations are explicitly defined based on a series of predefined semantic relationship and a set of three primitive ones (abstraction, reification, and translation). The transformation model could itself contain transformation rules.
4. The last principle, i.e., the standardization process, is only on the way. Only the future will tell us whether a wide adoption of the above techniques will be effective.

In addition to the adherence of the basic MDA principles, our UI engineering methodology classifies the involved models in a similar way. Fig. 7 graphically depicts the distribution of models according to the OMG paradigm of MDA and their UsiXML counterpart: task and domain models (T&C) are considered as the Computing Independent Model (CIM) as they are stated independently of any implementation of any interactive systems. Such models could be specified for virtually any UI type. The Platform Independent Model (PIM) is interpreted as the Abstract UI model (AUI) in UsiXML in the sense that it is independent of any interaction modality: at this level, we do not know yet whether the UI will be graphical, modal, virtual or multimodal. The Platform Specific Model (PSM) is interpreted as the Concrete UI model (CUI) in UsiXML in the sense that it is independent of any vocabulary of markup and programming languages. At this level, we have already chosen what kind on interaction modality will be exploited, but we do not know yet which physical computing platform will run the UI. This is why it should be believed that the CUI is not platform specific. Only some aspects of the target platform are selected, the platform being modeled itself in the platform model. In contrast with other MDA-compliant architectures, the present one can either render the UI directly (by interpretation) or automatically generate code (by generation) that will be compiled, linked with the rest of the application, and executed. Therefore, there is no 'model-to-code' transformation per se. Rather, different tools produce different results from the UsiXML specifications. For other 'model-to-model' transformations, graph transformation techniques are exploited throughout the development life cycle to maintain consistency.

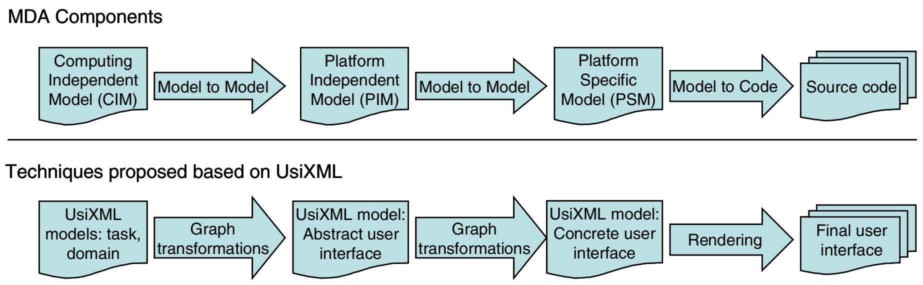

MDA Components
Computing Independent Model (CIM)
Model to Model
Platform Independent Model (PIM)
Model to Model
Platform Specific Model (PSM)
Model to Code
Source code
Techniques proposed based on UsiXML
UsiXML models: task, domain
Graph transformations
UsiXML model: Abstract user interface
Graph transformations
UsiXML model: Concrete user interface
Rendering
Final user interface

Fig. 7. The distribution of UsiXML models according to the MDA classification

5 Tools

Fig. 7 mainly depicts the forward engineering of UIs. As our UI engineering methodology is not restricted to this development path, other paths could be followed as well [13]. Fig. 8 somewhat generalizes the development life cycle, but for one context of use at a time. To support the application of a particular development path, a suite of tools has been developed and is currently being expanded (see http://www.usixml.org/index.php?view=page&idpage=20 for more information). The most significant tools belonging to this suite are (Fig. 8):

- TransformiXML is a Java-based application that is responsible for defining, storing, manipulating, and executing productions contained in graph grammars to support graph transformations (model-to-model transformations)
- IdealXML [16] is a Java-based application containing the graphical editor for the task model, the domain model, and the abstract model. It can also establish any mapping between these models either manually (by direct manipulation) or semi-automatically (by calling TransformiXML).
- KnowiXML [8] consists of an expert system based on Protégé that automatically produces several AUIs from a task and a domain models for various contexts.
- GrafiXML [12] is the most elaborate UsiXML high-fidelity editor with editing of the CUI, the context model and the relationships between. It is able to automatically generate UI code in HTML, XHTML, XUL and Java thanks to a series of plug-ins.

- VisiXML is a Microsoft Visio plug-in for drawing in mid-fidelity graphical UIs, that is UIs consisting exclusively of graphical CIOs. It then exports the UI definition in UsiXML at the CUI level to be edited by GrafiXML or another editor.
- SketchiXML [6] consists of a Java low-fidelity tool for sketching a UI for multiple users, multiple platforms (e.g., a Web browser, a PDA), and multiple contexts of use. It is implemented on top of Jack agent system.
- FormiXML is a Java editor dedicated to interactive forms with a smart system of copy/paste techniques to support reusability of components. It automatically generates the complete UI code for Java/Swing.
- Several renderers are currently being implemented: FlashiXML opens a CUI UsiXML file and renders it in Flash, QtkXML in the Tcl/Tk environment, and JaviXML for Java.
- VisualiXML [17] personalizes a UI and produces one thanks to generative programming techniques for Visual C++ V6.0.
- ReversiXML [2] opens a HTML file and reverse engineers it into UsiXML at both the CUI and AUI levels.

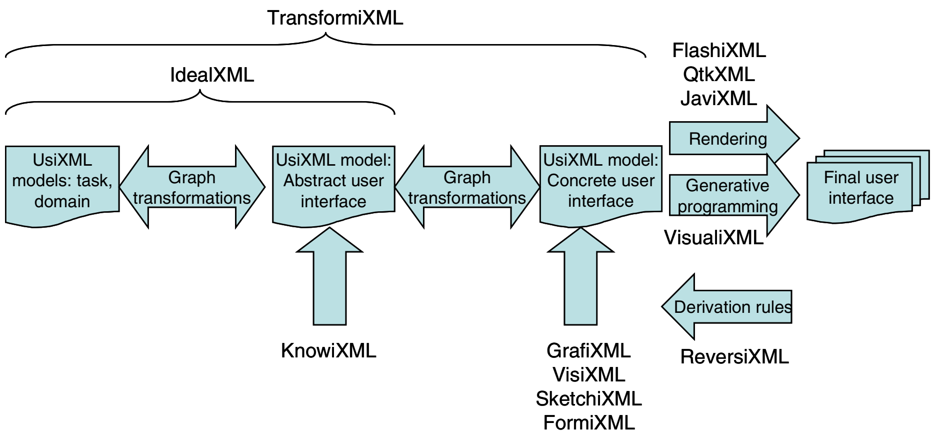


Fig. 8. The suite of UsiXML tools structured according to the MDA classification.

6 Conclusion

In this paper, we have introduced a UI Engineering methodology articulated on three axes: models and their specification language, method, and tools that support the method based on the underlying models. All aspects are stored in UsiXML (www.usixml.org) files that can be exchanged, shared, and communicated between stakeholders (designers, developers, and end users). It has been demonstrated that the global methodology adheres to the principles of MDA and is therefore compliant, except for the standardization process which is ongoing. It is worth to note that this environment has been largely studied and scrutinized for user interfaces of information systems that are equipped with different types of interfaces (graphical, vocal, virtual, and multimodal mainly) on different types of computing platforms. We believe that a significant portion of UsiXML models could be equally used for the UI of more sophisticated interactive systems (e.g., [9]), like safety-critical systems, industrial supervision systems, etc. But this is matter of more extensive study. It is likely that the model transformations will be more complex to discover and to apply.

Acknowledgements

The author would like to thank all members of the Belgian Lab. of Computer-Human Interaction (BCHI) for their fruitful involvement in the UsiXML initiative. We gratefully acknowledge the support of the following projects: the SIMILAR network of excellence (the European research task force creating human-machine interfaces similar to human-human communication – http://www.similar.cc), the SALAMANDRE research project (User Interfaces for Mobile and Multi-platform Interactive Systems, Initiatives III Research Program, DGTRE, Ministry of Walloon Region), the REQUEST research project (WIST-Wallonie Information Société Technologies program under convention n°031/5592), the Cameleon project (Context Aware Modelling for Enabling and Leveraging Effective interaction – FP6-IST5-2000, http://giove.cnuce.cnr.it/cameleon. html). We also thank very much our colleagues who have been involved in these projects for their fruitful exchanges and discussions.

References

1. Balme, L., Demeure, A., Barralon, N., Coutaz, J., Calvary, G.: CAMELEON-RT: a Software Architecture Reference Model for Distributed, Migratable, and Plastic User Interfaces. In: Proc. of EUSAI'2004. Lecture Notes in Computer Science, Vol. 3295. Springer-Verlag, Berlin (2004) 291–302
2. Bouillon, L., Vanderdonckt, J., Chow, K.C.: Flexible Re-engineering of Web Sites; In: Proc. of 8th ACM Int. Conf. on Intelligent User Interfaces IUI'2004 (Funchal, 13-16 January 2004). ACM Press, New York (2004) 132–139
3. 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) 289–308
4. Constantine, L.L.: Canonical Abstract Prototypes for Abstract Visual and Interaction Design. In: Proc. of 10th Int. Workshop on Design, Specification, and Verification of Interactive Systems DSV-IS'2003 (Funchal, June 11-13, 2003). Lecture Notes in Computer Science, Vol. 2844. Springer-Verlag, Berlin (2003) 1–15. Accessible at http://www.foruse. com/articles/abstract.pdf
5. Coutaz, J.: PAC, an Object Oriented Model for Dialog Design. In: Proc. of 2nd IFIP International Conference on Human-Computer Interaction Interact'87 (Stuttgart, September 1-4, 1987). North Holland, Amsterdam (1987) 431–436.
6. Coyette, A., Vanderdonckt, J.: A Sketching Tool for Designing Anyuser, Anyplatform, Anywhere User Interfaces. In: Proc. of 10th IFIP TC 13 Int. Conf. on Human-Computer Interaction INTERACT'2005 (Rome, 12-16 September 2005). Lecture Notes in Computer Science. Springer-Verlag, Berlin (2005)

7. 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) 140–147
8. Furtado, E., Furtado, V., Soares Sousa, K., Vanderdonckt, J., Limbourg, Q.: KnowiXML: A Knowledge-Based System Generating Multiple Abstract User Interfaces in UsiXML. In: Proc. of 3rd Int. Workshop on Task Models and Diagrams for user interface design TAMODIA'2004 (Prague, November 15-16, 2004). ACM Press, New York (2004) 121–128
9. Grolaux, D., Van Roy, P., Vanderdonckt, J.: Migratable User Interfaces: Beyond Migratory User Interfaces. In: Proc. of 1st IEEE-ACM Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services MOBIQUITOUS'04 (Boston, August 22-25, 2004). IEEE Computer Society Press, Los Alamitos (2004) 422–430
10. Jacob, R., Limbourg, Q., Vanderdonckt, J.: Computer-Aided Design of User Interfaces IV. Proc. of 5th Int. Conf. of Computer-Aided Design of User Interfaces CADUI'2004 (Funchal, 13-16 January 2004). Information Systems Series, Kluwer Academics, Dordrecht (2005)
11. Kolski, Ch., Vanderdonckt, J.: Computer-Aided Design of User Interfaces III. Proc. of 4th Int. Conf. of Computer-Aided Design of User Interfaces CADUI'2002 (Valenciennes, 15-17 May 2002). Information Systems Series, Kluwer Academics, Dordrecht (2002)
12. Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., Lopez, V.: UsiXML: a Language Supporting Multi-Path Development of User Interfaces. In: Proc. of 9th IFIP Working Conference on Engineering for Human-Computer Interaction jointly with 11th Int. Workshop on Design, Specification, and Verification of Interactive Systems EHCI-DSVIS'2004 (Hamburg, July 11-13, 2004). Springer-Verlag, Berlin (2005)
13. Limbourg, Q., Multi-path Development of User Interfaces. Ph.D. thesis. Université catholique de Louvain, IAG-School of Management. Louvain-la-Neuve (Nov. 2004).
14. Mellor, S.J., Scott, K., Uhl, A., Weise, D.: MDA Distilled: Principles of Model-Driven Architecture. Addison-Wesley, New York (2004)
15. Molina, J.P., Vanderdonckt, J., Montero, F., Gonzalez, P.: Towards Virtualization of User Interfaces. In: Proc. of 10th ACM Int. Conf. on 3D Web Technology Web3D'2005 (Bangor, March 29-April 1, 2005). ACM Press, New York (2005)
16. Montero, F., Lozano, M., González, P.: IDEALXML: an Experience-Based Environment for User Interface Design and pattern manipulation. Technical Report DIAB-05-01-4. Universidad de Castilla-La Mancha, Albacete (2005).
17. Schlee, M., Vanderdonckt, J.: Generative Programming of Graphical User Interfaces. In: Proc. of 7th Int. Working Conference on Advanced Visual Interfaces AVI'2004 (Gallipoli, May 25-28, 2004). ACM Press, New York (2004) 403–406
18. Vanderdonckt, J., Bouillon, L., Chieu, K.C., Trevisan, D.: Model-based Design, Generation, and Evaluation of Virtual User Interfaces. In: Proc. of 9th ACM Int. Conf. on 3D Web Tech. Web3D'2004 (Monterey, April 5-8, 2004). ACM Press, New York (2004) 51–60
19. 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) 424–429
20. Vanderdonckt, J., Puerta, A.R.: Computer-Aided Design of User Interfaces II. Proc. of 3rd Int. Conf. of Computer-Aided Design of User Interfaces CADUI'99 (Louvain-la-Neuve, 21-23 October 1999). Information Systems Series, Kluwer Academics, Dordrecht (1999)
21. Vanderdonckt, J.: Computer-Aided Design of User Interfaces. Proc. of 2nd Int. Workshop on Computer-Aided Design of User Interfaces CADUI'96 (Namur, 5-7 June 1996). Collection « Travaux de l'Institut d'Informatique » n°15. Presses Universitaires de Namur, Namur (1996)