# USIXML: A USER INTERFACE DESCRIPTION LANGUAGE SUPPORTING MULTIPLE LEVELS OF INDEPENDENCE

QUENTIN LIMBOURG and JEAN VANDERDONCKT

*Université catholique de Louvain, School of Management (IAG)*
*Place des Doyens, 1 – B-1348 Louvain-la-Neuve, Belgium*
*{limbourg, vanderdonckt} @ isys.ucl.ac.be – http://www.usixml.org*

USer Interface eXtensible Markup Language (UsiXML) consists of a User Interface Description Language (UIDL) allowing designers to specify a user interface at multiple levels of abstraction depending on the development path they are following: task and concepts, abstract user interface, concrete user interface, and final user interface. These levels support to some extent independence with respect to device, computing platform, modality of interaction, channel of information, and context of use. A single user interface can be specified and produced at and from different, possibly multiple, levels of abstraction while maintaining the mappings between these levels if required. Thus, the development process can be initiated from any level of abstraction and proceed towards obtaining one or many final user interfaces for various contexts of use (forward engineering), by recovering the final user interface into any upper level (reverse engineering), or by adapting at any level of abstraction (reengineering).

## 1 Introduction

Due to the rapid changes of today's organisations and their business, many information systems departments face the problem of quickly adapting the User Interface (UI) of their interactive applications to these changes. These changes include, but are not limited to: task redefinition or reallocation among workers [4], support of new computing platforms, migration from stationary platforms to mobile computing [16], evolution of users with more demands, increasing need for more usable UIs, transfer of task from one user to another one [6], redefinition of the organisation structure, adaptation to dynamic environments [15], change of the language, redesign due to obsolescence [3], evolution of the domain model [1]. All these changes change to some extent the context of use, which is hereby referred to as the complete environment where final users have to carry out their interactive tasks to fulfil the roles they are playing in their organisations.

Organisations react to changes in different ways in their UI development processes. As a result, different *development scenarios* may be identified as potential development practices adopted in reaction to change. For instance, one organisation starts by recovering existing input/output screens, by redrawing them and by completing the functional core when the new UI is validated by the customer (*reverse engineering approach*). Another organisation prefers to modify the domain model (e.g., a UML class diagram [11]) and the task model [17] to be mapped further to

screen design (*forward engineering approach*). A third one tends to apply in parallel all the required adaptations where they occur (*wide spreading approach*). A fourth one relies on an intermediate model and propagates changes to all artefacts exploited in the development process (*middle-out approach*) [14]. The UI development process has also been empirically observed as an ill-defined, incomplete, and incremental process [21] that is not well supported by rigid development methods and tools. Existing methods and tools usually force developers to act in a pre-defined and inflexible way. The variety of the approaches adopted in organisations and the rigidity of existing solutions provide ample motivations for a UI development paradigm that is flexible enough to accommodate multiple development paths and design situations while staying precise enough in the definition of its concepts and development steps. To overcome these shortcomings, the development paradigm of **multi-path UI development** is characterised by the following principles:

- *Formal definition of UI models*: any UI is expressed through to a suite of models that are analysable, editable, and exploitable by software.
- *Transformational approach*: each model stored according to the ontological format can be subject to transformations realizing various development steps.
- *Multiple development paths*: development steps can be combined together to form development paths that are compatible with the organisation's development scenario. For example, a series of transformations can be applied to progressively move from a task model to a dialog model, to recover a domain model from a presentation model, or to derive a presentation model from both the task and domain models.
- *Flexible development approaches*: development scenarios (e.g., forward engineering, reverse engineering, wide spreading, or middle-out) are supported by flexibly following alternate development paths.

The remainder of this paper is structured as follows: Section 2 reports on some significant work related to the multi-path UI development. Section 3 introduces the reference representations used throughout this paper to address the principles of expressiveness and central storage of models based on USer Interface eXtensible Markup Language (UsiXML). Section 4 shows how a transformational approach is represented and implemented thanks to graph grammars and graph transformations applied on model expressed in UsiXML and stored in the model repository. Section 5 provides an overview of tools developed so far around UsiXML. Section 6 concludes the paper by reporting on some benefits and difficulties encountered so far.

## 2 Related Work

The multi-path UI development, as defined in Section 1, is at the intersection of two areas of research and development: on the one hand, UI modelling and design of multi-platform UIs represent significant advances in Human-Computer Interaction

(HCI) and on the other hand, program transformation is considered promising in Software Engineering (SE) and as a mean to bridge the gap between [4].

Teallach tool and method [10] exploit three models: a task model, a domain model as a class diagram, and a presentation model both at logical and physical levels. Teallach enables designers to start building a UI from any model and maps concepts from different models one to each other.

UIML consists of a UI Description Language (UIDL) supporting the development of UIs for multiple computing platforms by introducing a description that is platform-independent to be further expanded with peers once a target platform has been chosen [2]. The TIDE tool [2] transforms a basic task model into a final UI.

XIML [8] is a more general UIDL than UIML as it can specify any type of model, any model element, and relationships between. The predefined models and relationships can be expanded to fit a particular context of use.

SeescoaXML [15] is the UIDL exploited in the SEESCOA project to support the production of UIs for multiple platforms and the run-time migration of the full UI.

TERESA (Transformation Environment for inteRactivE Systems representAtions) [16] producing different UIs for multiple computing platforms by refining a general task model for the different platforms. Then, various presentation and dialogue techniques are used to map the refinements into XHTML code adapted for each platform such as Web, PocketPC, and mobile phones.

RIML [19] consists of an XML-based language combines features of several existing markup languages (e.g., XForms, SMIL) in a XHTML language profile. This language is used to transform any RIML-compliant document into multiple target languages suitable for visual or vocal browsers on mobile devices.

The above pieces of work all represent an instance with some degree of coverage and restrictions of the multi-path UI development. Regarding the *UI expressiveness* for multiple contexts of use, UIML, RIML, XIML, TERESA and SeescoaXML are UIDLs that address the basic requirements of UI modelling and expressivity. XIML is probably the most expressive one as a new model, element or relationship can be defined internally. Yet, there is no systematic support of these relationships until they are covered by specific software. Regarding the *transformational approach*, Seescoa, Teallach, TERESA and TIDE include some transformation mechanism to map a model onto another one, but the logics and the definition of transformation rules are completely hard coded with little or no control by designers. In addition, the definition of these representations is not independent of the transformation engine. Regarding multiple development paths, only Teallach explicitly addresses the problem as models can be mapped one onto another according to different ways. Other typically apply top-down (e.g., TIDE), bottom-up (e.g., VAQUITA [3]), middle-out (e.g., MIDAS [14]), but none of them support all approaches.

To satisfy the requirements subsumed by the four principles of multi-path development, *Graph Transformation* (GT) [18] will be exploited as a mean to realize development steps. Substantive experience shows applicability in numerous fields of science (e.g., biology, operational research) and, notably, to computer science (e.g., model checking, parallel computing, software engineering). GTs are operated in two steps: expressing abstract concepts in the form of a graph structure and defining operations producing relevant transformations on the graph structure. Sucrow [20] used GT techniques to formally describe UI dialog with dialog states (the appearance of a UI at a particular moment in time) and dialog transitions (transformations of dialog states). To support "a continuous specification process of graphical UIs", two models are defined in the development process: abstract and concrete. Elements such as dialog patterns, style guides, and metaphors are used to automate abstract to concrete transitions. However, conceptual coverage and fundamental aspects of this work remain missing: presented concepts are limited at the model level without going to any final UI and there is no description of the ontology underlying the method.

To structure the models involved in the UI development process and to characterise the model transformations to be expressed through GTs, a reference framework is now introduced.

## 3 The Reference Framework used for Multi-path UI Development

Multi-path UI development is based on the Cameleon Reference Framework [5] 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 artefact of interest as a model or a UI representation) [5]:

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).
2. *Concrete UI* (CUI): concretises an abstract UI for a given context of use into Concrete Interaction Objects (CIOs) so as to define widgets layout and interface navigation. This process 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. A CUI is also an abstraction of the FUI with respect to the platform.
3. *Abstract UI* (AUI): defines interaction spaces (or presentation units) by grouping subtasks according to various criteria (e.g., task model structural patterns, cognitive load analysis, semantic relationships identification), a navigation scheme between the interaction spaces and selects Abstract Interaction Objects (AIOs) for each concept so that they are independent of any context of use. An AUI abstracts a CUI into a UI definition that is independent of any modality of

interaction (e.g., graphical interaction, vocal interaction). 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.

4. *Task & Concepts* (T&C): describe the various tasks to be carried out and the domain-oriented concepts as they are required by these tasks to be performed. These objects are instances of classes representing the concepts manipulated.

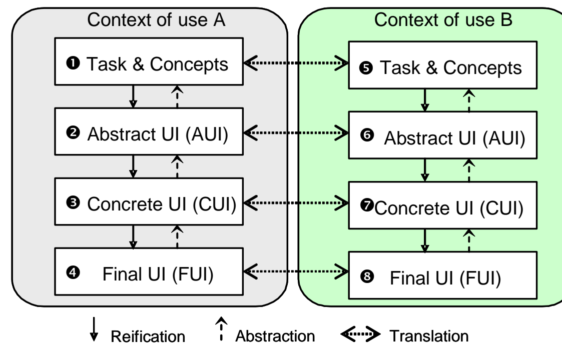


Figure 1 The Cameleon Reference Framework

This framework exhibits three types of *basic transformation types*: (1,2) *Abstraction* (respectively, *Reification*) is a process of elicitation of artefacts that are more abstract (respectively, concrete) than the artefacts that serve as input to this process. Abstraction is the opposite of reification. (3) *Translation* is a process that elicits artefacts intended for a particular context of use from artefacts of a similar development step but aimed at a different context of use. With respect to this framework, multi-path UI development 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 all basic transformation types and their combinations of Fig. 1. To enable such a development, two important requirements gathered from observations are:

1. A language that enables the expression and the manipulation (e.g., creation, modification, deletion) of the model at each development steps and for each context of use. For this purpose, UsiXML is introduced and defined.
2. A mechanism to express design knowledge that would provide a substantial support to the designer in the realisation of transformations. For this purpose, a GT technique is introduced and defined based on UsiXML.

## 4 Graph Transformation Specification with UsiXML

Graph transformation techniques were chosen to formalize UsiXML, the language designed to support multi-path UI development, 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 graph transformation 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 UsiXML (Fig. 2). UsiXML model collection is structured according to the four basic levels of abstractions defined in the Cameleon Reference Framework that is intended to express the UI development life cycle for context-sensitive interactive applications. Each level of Fig. 1 can be itself further decomposed into two sub-levels (Fig. 2):

- At the FUI level, the rendering materialises how a particular UI coded in one language (markup, programming or declarative) is rendered depending on the UI toolkit, the window manager and the presentation manager. For example, a push button programmed in HTML at the code sub-level can be rendered differently, here on MacOS X and Java Swing. Therefore, the code sub-level is materialised onto the rendering sub-level.
- Since the CUI level is supposed to abstract the FUI independently of any computing platform, this level can be further decomposed into two sub-levels: platform-independent CIO and CIO type. For example, a HTML push-button belongs to the type "Graphical 2D push button". Other members of this category include a Windows push button and XmButton, the OSF/Motif counterpart.
- Since the AUI level is assumed to abstract the CUI independently of any modality of interaction, this level can be further decomposed into two sub-levels: modality-independent AIO and AIO type. For example, a software control (whether in 2D or in 3D) and a physical control (e.g., a physical button on a control panel or a function key) both belong to the category of control AIO.
- At the T&C level, a task of a certain type (here, download a file) is specified that naturally leads to AIO for controlling the downloading.

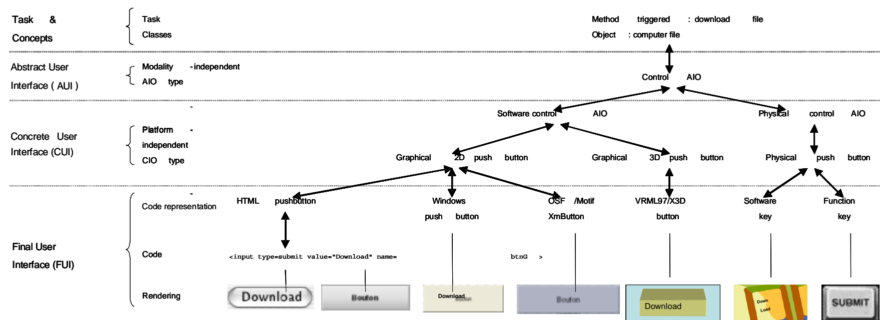


Figure 2 Example of transformations in UsiXML

The organisation of the framework into four levels allows any transformation of a UI model from one level to another one. Furthermore, transformations can be combined so as to define more complex transformations. For example, if a Graphical

User Interface (GUI) needs to be virtualised, a series of abstractions is applied until the sub-level "Software control AIO" sub-level is reached. Then, a series of reifications can be applied to come back to the FUI level to find out another object satisfying the same constraints, but in 3D. If the GUI needs to be transformed for a UI for augmented reality for instance, the next sub-level can be reached with an additional abstraction and so forth.

Combining transformations allows establishing development paths. Here, some first examples are given of multi-path UI development. To face multi-path development of UIs in general, UsiXML is equipped with a collection of basic UI models (i.e., domain model, task model, AUI model, CUI model, context model and mapping model) (Fig. 4) and a so-called *transformation model* (Fig. 3) [12].

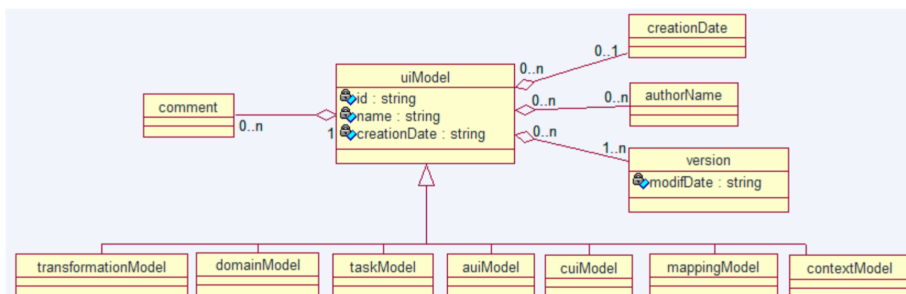


Figure 3 UsiXML Model Collection

Beyond the task and domain models, the AUI and CUI models, UsiXML is equipped with the following models:

- 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.
- 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.
- 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. Consequently, a context model consists of a user model, a platform model, and an environment model.
- transformationModel: is a model containing a description of model transformations as defined in the framework in fig. 1.

Transformations are specified using transformation systems. Transformation systems rely on the theory of graph grammars [18]. We first explain what a transformation system is and then illustrate how they may be used to specify UI model transformation. The proposed formalism to represent model-to-model transformation in UsiXML is graph transformations. This formalism has been discussed in [12,13]. UsiXML has been designed with an underlying graph structure. Consequently any graph transformation rule can be applied to a UsiXML specification. A *transformation system* is composed of several transformation rules. Technically, a rule is graph rewriting rule equipped with negative application conditions and attribute conditions [19].

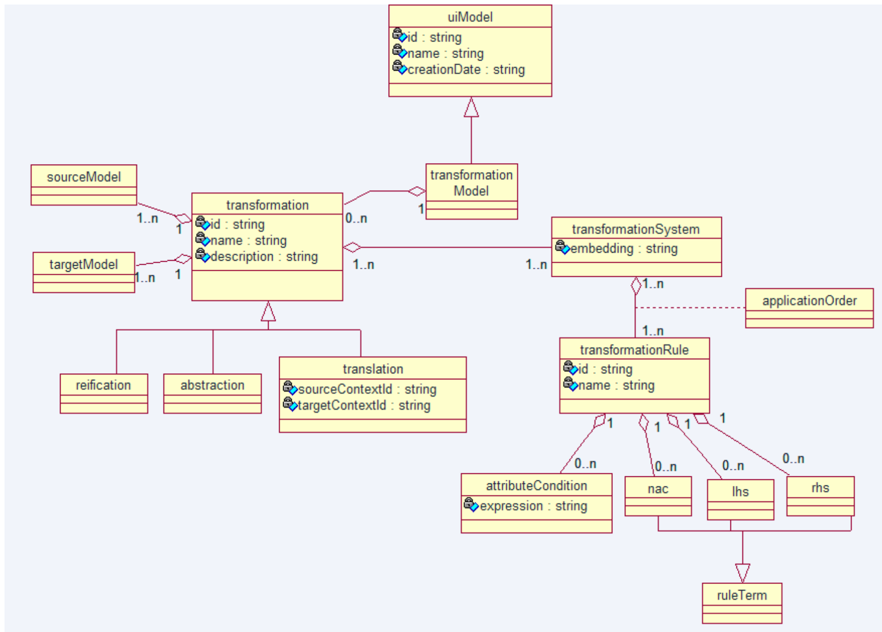


Figure 4 Transformation model as defined in UsiXML

Fig. 5 illustrates how a transformation system applies to a UsiXML specification: let G be a UsiXML specification, when 1) a *Left Hand Side* (LHS) matches into G and 2) a *Negative Application Condition* (NAC) does not matches into G (note that several NAC may be associated with a rule) 3) the LHS is replaced by a *Right Hand Side* (RHS). G is resultantly transformed into G, a resultant UsiXML specification. All elements of G not covered by the match are considered as unchanged. All elements contained in the LHS and not contained in the RHS are considered as deleted (i.e., rules have destructive power). To add to the expressive power of transformation rules, variables may be associated to attributes within a LHS. Theses variables are initialised in the LHS, their value can be used to assign an attribute in the expression of the RHS (e.g., LHS: button.name:=x, RHS : task.name:=x). An ex-

pression may also be de-fined to compare a variable declared in the LHS with a constant or with another variable. This mechanism is called *attribute condition*. We detail hereafter a simplified scenario illustrating the three basic types of transformation (thus inducing directionalities) mentioned in Section 3.

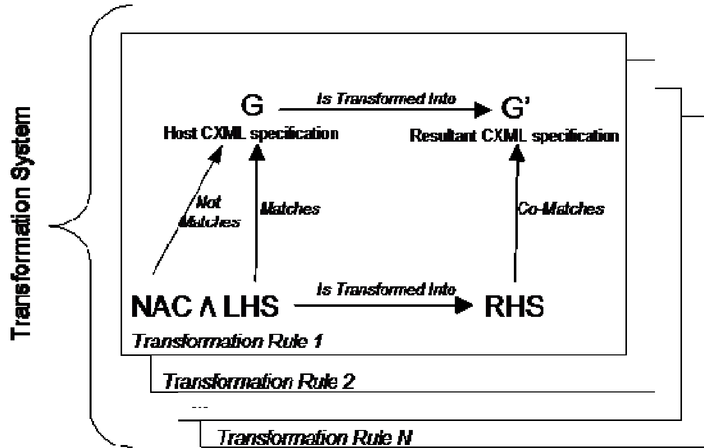


Figure 5 Transformation system in UsiXML

**Step 1-Abstraction**: a designer reverse engineers an HTML page with ReversiXML [3] in order to obtain a CUI model. Transformation 1 (Fig. 6) is an abstraction that takes a button at the concrete level and abstracts it away into an abstract interaction object. The LHS selects every button and the method they activate and create a corresponding abstract interaction object equipped with a control facet mapped onto the method triggered by its corresponding concrete interaction object. Some behavioural specification is preserved at the abstract level.

Note that behaviour specification in UsiXML is also done with graph transformations rules. It is out of the scope of this paper to explicit this mechanism. This is why rule 1 in transformation 1, in its LHS, embeds a fragment of transformation system specification. This may seem confusing at first sight but is very powerful at the end i.e., we dispose of a mechanism trans-forming a UI behavioural specification into another one! In the RHS, one also see that a relationship is abstracted into has been created. This relationship ensures traceability of rule application and helps in maintaining coherence among different levels of abstraction.

**Step 2-Reification** : the designer decides to add, by hand, to the abstract level a navigation facet to every abstract interaction object that has a control facet. From this new abstract specification, Transformation 2 (Fig. 7) reifies every abstract interaction object into image components (i.e., a type of CIO). By default, the control facet is activated when an event onMouseOver is triggered, and the navigation facet is activated when the imageComponent is double-clicked. This rule may be customized by the designer to reflect his own preference or needs.

```xml
<abstraction id="AB1" name = "AbstractButtonWithCon-trol"
description = "this translation abstracts buttons into
an AIO with an activation facet">
<transformationSystem id = "TR2" name="Transfo2"...>
<transformationRule id = "rule1" name "abstractsBut">
<lhs>
  <button ruleSpecificID="1" mapID="2">
    <behavior>
      <action>
      <transformationSystem>
        <transformationRule>
          <rhs>
            <method ruleSpecificID="3"  mapID ="4" name="X" />
              <isTriggeredBy isFired="true">
                <source sourceId="1">
                <target targetId="3">
              </isTriggeredBy>
          </rhs>
        </transformationRule>
      </transformationSystem>
      </action>
    </behavior>
  </button>
</lhs>
<rhs>
  <abstractIndividualComponent ruleSpecificId="5">
    <control activatedMethod="X">
  </abstractIndividualComponent>
  <isAbstractedInto>
    <source sourceId="2"/>
    <target targetId="5"/>
  <isAbstractedInto>
  <button ruleSpecificId="1" mapID="2">
    <behavior>
      <transformationSystem>
        <transformationRule>
          <rhs>
            <method ruleSpecificID="3" mapID ="4"/>
              <isTriggeredBy isFired="true">
                <source sourceId="1">
                <target targetId="3">
              </isTriggeredBy>
          </rhs>
        </transformationRule>
      </transformationSystem>
    </behavior>
  </button>
</rhs>
<nac.../>
</transformationRule>
</transformationSystem>
</abstraction>
```

Figure 6 An abstraction

```xml
 <reification id="Reif1" name = "ReifiesAioImgCtlrNav"
description = " reifies a control AIO into an image
component with corresponding behavior template">
<transformationSystem id = "TRE1" name="TR2"...>
  <transformationRule id = "rule44" name "ReiFControl44">
    <lhs>
      <abstractIndividualComponent mapID="1">
        <control activatedMethod="X"/>
        <navigation target="Y"/>
      </abstractIndividualComponent>
    <lhs>
    <rhs>
      <imageComponent ruleSpecificID="2">
        <behavior>
          <event type="doubleClick"/>
          <action>
            <transformationSystem>
              <transformationRule>
                <lhs/>
                <rhs>
                  <method ruleSpecificID="3" name="X"/>
                  <isTriggeredBy isFired="true">
                    <source sourceId="2">
                    <target targetId="3">
                  </isTriggeredBy>
                </rhs>
              </transformationRule>
            </transformationSystem>
          </behavior>
        <behavior>
          <event type="onMouseOver(self)"/>
          <action>
            <transformationSystem>
              <transformationRule>
                <lhs/>
                <rhs>
                  <graphicalContainer id="Y" visible="true"/>
                </rhs>
              </transformationRule>
            </transformationSystem>
        </behavior>
      </imageComponent>
      <isReifiedInto>
        <source sourceId="1"/>
        <target targetId="2"/>
      </isReifiedInto>
      <abstractIndividualComponent mapID="1">
        <control activatedMethod="X">
      </abstractIndividualComponent>
    </rhs>
    <nac.../>
  </transformationRule>
</transformationSystem>
</reification>
```

Figure 7 A reification

**Step 3-Translation**: to adapt a UI to a new type of display/browser that has the characteristic to be tall and narrow. The designer decides then to apply Transformation 3 (Fig. 8) to her CUI model. This transformation is made of a rule that rule selects all boxes (basic layout structure at the CUI level), sets these boxes to vertical. All widgets contained in this box are then glued to the left of the box (again in the idea of minimizing the width of the resulting UI). Note the presence of a negative application condition (too long to show in previous examples) that ensures that rule 1 in transformation 3 is not applied to an already formatted box. Fig. 8 shows a simple example of translation specified with UsiXML. This rule of the rule selects all boxes (basic layout structure at the CUI level), sets these boxes to vertical. All wid-

gets contained in this box are then glued to the left of the box (again in the idea of minimizing the width of the resulting UI). A negative application condition ensures that a rule is not applied to an already formatted box.

```
<translation id="TL1" name="squeezeDisplay"
description= "this translations vertically aligns all widgets of a con-tainer">
   <sourceModel type="cui"/>
   <targetModel type="cui"/>
<transformationSystem id="TR1" name="Transfo1"...>
   <transformationRule id="rule1" name="squeeze1">
      <lhs>
         <box mapID="1">
            <graphicalIndividualComponent mapId="2" />
         </box>
      </lhs>
      <rhs>
         <box mapID="1" type="vertical">
         <graphicalIndividualComponent mapId="2" glueHorizontal="left"/>
         </box>
      </rhs>
      <nac>
         <box mapID="1" type="vertical">
         <graphicalIndividualComponent mapId="2" glueHorizontal="left"/>
      </nac>
   </transformationRule>
</transformationSystem>
</translation>
```

Figure 8 Transformation 3

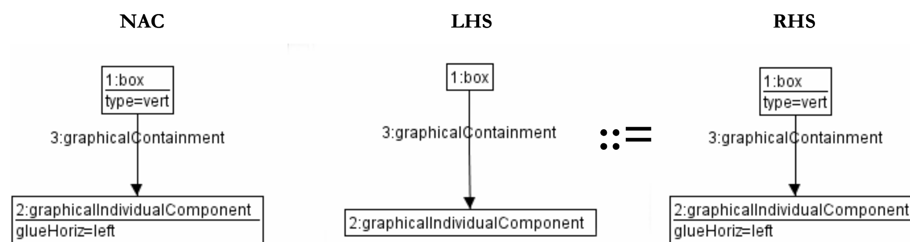


Figure 9 Graphical representation of the transformation

Alternatively to textual representation, transformation rules are easily expressed in a graphical syntax. Fig. 9 shows a graphical equivalent of the rule contained in Fig. 8. A general purpose tool for graph transformation called AGG (Attributed Graph Grammars) was used to specify this example. No proof the superiority of graphical formalism over textual ones, but at least UsiXML designer has the choice between both. *Traceability* (and as a side-effect reversibility) of model transformation is enabled thanks to a set of 'so-called' interModelMappings (e.g., isAbstractedInto, IsReifiedInto, isTranslatedInto) allowing to relate model elements belonging to different models. As so it is possible to keep a trace of the application of rules i.e., when a new element is created a mapping indicates of what element it is an abstraction, a reification, a translation, etc. Another advantage of using these mappings is to support multi-path development is that they explicitly connect the various levels of our framework and realizes a seamless integration of the different models used to describe the system. Knowing the mappings of a model increases dramatically the understanding of the underlying structure of a UI. It enables to answer, at no cost, to

question like: what task a interaction object enables?, what domain object attributes are updated by what interaction object?

## 5 Tool Support

Table 1 provides an overview of major tools developed around UsiXML. Some tools are dedicated to editing various models involved in UsiXML while others are devoted to producing (semi-)automatically models and code from and to the four different levels of the reference framework depicted in fig. 1.

| | |
|---|---|
| GrafiXML* | Graphical model editor: CUI (high fidelity), context model + Textual model editor: all UsiXML models + Code generation: XHTML 1.0, Java Swing |
| VisiXML* | Graphical model editor: CUI (mid fidelity) |
| SketchiXML [7] | Model editor: CUI (pen-based sketching tool) |
| IdealXML* | Model editor: Task & Domain, AUI, inter-model relationships |
| KnowiXML [9] | Model editor: task & AUI + Model transformation: from Task to AUI |
| ReversiXML*[3] | Reverse engineering: from HTML 4.0 to CUI and/or AUI |
| TransformiXML* | Model transformation: from any UsiXML model to any UsiXML model |

*See UsiXML web site at http://www.usixml.org

Table 1 Set of tools currently developed around UsiXML

## 6 Conclusion

Information systems are subject to a constant pressure toward change and adaptation to ever changing contexts of work. UIs represent an important and a crucial software component of information system. Multi-path UI development has been proposed to cope with the problem of UI adaptation to an evolving context of use. Multi-path UI development has been defined as an engineering method and tool that enables a designer to start a UI development by several entry points in the development cycle and from this entry point get a substantial support to build a qualitative UI. Main features of multi-way UI development are:

1. A flexible development process based on transformations
2. A unique formal language to specify UI related artefacts. So far, these concepts have been hard coded in software tools, thus preventing anyone from reusing, re-defining or exchanging them. UsiXML provides a mean to overcome these shortcomings. The core of this language is composed of a set of integrated models expressed in a formal and uniform format, governed by a common meta-model definition, graphically expressible and a modular, modifiable and extensible repository of executable design knowledge that is also represented with a

graphical syntax. Furthermore, a definition of an XML notation supporting the exchange of models and executable design knowledge has been presented.

3. A transformational approach based on systematic rules that guarantee semantic equivalence when applied, some of them being reversible.
4. A tool supporting the expression and manipulation of models and design knowledge visually.

With increase of design experience, a copious catalogue of transformation rules can be assembled into meaningful grammars. The level of support provided to the accomplishment of design steps varies from one transition to another. Indeed, some transitions are better known than others. For instance, the reification between physical and logical UI can be supported by hundreds of rules namely by widget selection rules. On the contrary, rules that enable the translation of a task model from a desktop PC to a handheld PC are, for now, understudied. Some transitions are intrinsically harder to support (e.g., abstraction transitions). For instance, retrieving a task model from the physical UI is not a trivial problem.

**Acknowledgements**

We hereby acknowledge the support of the Cameleon research project (http://giove.cnuce.cnr.it/cameleon.html) under the umbrella of the European Fifth Framework Programme (FP5-2000-IST2) and of the SIMILAR network of excellence (http://www.similar.cc), the European research task force creating human-machine interfaces similar to human-human communication of the European Sixth Framework Programme (FP6-2002-IST1-507609).

**References**

1. Agrawal, A., Karsai, G. and Ledeczi, K., An End-to-end Domain-Driven Software Development Framework. in OOPSLA'2003: Companion of the 18th Annual ACM SIGPLAN Conf. on Object-oriented Programming Systems, Languages, and Applications. ACM Press, New York, 2003, 8–15.
2. Ali, M.F., Pérez-Quiñones, M.A. and Abrams M., Building Multi-Platform User Interfaces with UIML. in Multiple User Interfaces: Engineering and Application Framework. John Wiley and Sons, New York, 2003.
3. Bouillon, L., Vanderdonckt, J. and Chow, K.C., Flexible Re-engineering of Web Sites. in Proc. of 8th ACM Int. Conf. on Intelligent User Interfaces. 132–139.
4. Brown, J., Exploring Human-Computer Interaction and Software Engineering Methodologies for the Creation of Interactive Software. SIGCHI Bulletin, 29 (1), 1997, 32–35.
5. Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L. and Vanderdonckt, J., A Unifying Reference Framework for Multi-Target User Interfaces. Interacting with Computers, 15(3), 2003, 289–308.
6. Chikofsky, E.J. and Cross, J.H., Reverse Engineering and Design Recovery: A Taxonomy. IEEE Software, 1 (7), 1990, 13–17.

7. Coyette, A., Faulkner, S., Kolp, M., Limbourg, Q., Vanderdonckt, J., Sketchi-XML: Towards a Multi-Agent Design Tool for Sketching User Interfaces Based on UsiXML. in Proc. of Tamodia'2004
8. Eisenstein, J., Vanderdonckt, J. and Puerta, A., Model-Based User-Interface Development Techniques for Mobile Computing. in Proc. of 5th ACM Int. Conf. on Intelligent User Interfaces IUI'2001. ACM Press, New York, 2001, 69–76.
9. Furtado, E., Furtado, V., Sousa, K., Vanderdonckt, J., Limbourg, Q., Knowi-XML: A Knowledge-Based System Generating Multiple Abstract User Interfaces in UsiXML. in Proc. of Tamodia'2004
10. Griffiths, T., Barclay, P.J., Paton, N.W., McKirdy, J., Kennedy, J., Gray, P.D., Cooper, R., Goble, C.A. and da Silva, P.P., Teallach: A Model-Based User Interface Development Environment for Object Databases. Interacting with Computers, 14(1), December 2001, 31–68.
11. Larman, C., Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process. Prentice Hall, 2001.
12. Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L. and Lopez, V., UsiXML: a Language Supporting Multi-Path Development of User Interfaces. in Proc. of 9th IFIP Working Conf. on Engineering for Human-Computer Interaction jointly with 11th Int. Workshop on Design, Specification, and Verification of Interactive Systems. Kluwer Academics, Dordrecht, 2004.
13. Limbourg, Q. and Vanderdonckt, J., Transformational Development of User Interfaces with Graph Transformations. in Proc. of 5th Int. Conf. on Computer-Aided Design of User Interfaces. Kluwer Academics Pub., 2004, pp. 107-120.
14. Luo, P., A Human-Computer Collaboration Paradgim for Bridging Besign Conceptualization and Implementation. in Proc. of DSV-IS'94. 129–147.
15. Luyten, K., Van Laerhoven, T., Coninx, K. and Van Reeth, F.: Runtime Transformations for Modal Independent User Interface Migration. Interacting with Computers, 15(3), 2003, 329–347.
16. Mori, G., Paternò, F. and Santoro, C., Tool Support for Designing Nomadic Applications. in Proc. of 7th ACM Int. Conf. on Intelligent User Interfaces. ACM Press, New York, 2003, 141–148.
17. Paternò, F., Model-Based Design and Evaluation of Interactive Applications. Springer-Verlag, Berlin, 2000.
18. Rozenberg, G. ed. Handbook of Graph Grammars and Computing by Graph Transformation. World Scientific, Singapore, 1997.
19. Spriestersbach, A., Ziegert, T., Grassel, G., Wasmund, M., Dermler, G., A Single source authoring language to enhance the access from mobile devices to Web enterprise applications, WWW2003 Developers Day Mobile Web Track, 12th World Wide Web Conference WWW'03 (Budapest, 20-24 May 2003).
20. Sucrow, B., On Integrating Software-Ergonomic Aspects in the Specification Process of Graphical User Interfaces. Transactions of the SDPS Journal of Integrated Design & Process Science. 2(2), June 1998, 32–42.
21. Sumner, T., Bonnardel, N. and Kallak, B.H., The Cognitive Ergonomics of Knowledge-Based Design Support Systems. in Proc. of CHI'97, 83–90.