A Comparative Analysis of Transformation Engines for User Interface Development

Juan Manuel González Calleros¹, Adrian Stanciulescu¹, Jean Vanderdonckt¹, Jean-Pierre Delacre¹, Marco Winckler¹,²

¹Université catholique de Louvain, Louvain School of Management (LSM)
Place des Doyens, 1 – B-1348 Louvain-la-Neuve (Belgium).
E-mail: {juan.gonzalez@, jean.vanderdonckt@, adrian.stanciulescu@}uclouvain.be
Web: http://www.isys.ucl.ac.be/bchi/ – Tel: +32 10478349, 8525 – Fax: +32 10 478324
²LIHIS-IRIT, Université Paul Sabatier
118 route de Narbonne, Toulouse F-31062 (France)
E-mail: winckler@isys.ucl.ac.be

Abstract. In software engineering, transformational development aims at developing software systems by transforming a coarse-grained specification to final code through a sequence of small transformation steps. This transformational development method has followed a long tradition of establishing models and maintaining mappings between them so as to create and maintain accurate specifications of a user interface. Although several transformation engines support these mappings, one of the major difficulties lies in the accurate selection of a transformation engine capable of support a transformational approach for ensuring model-driven engineering of user interfaces. This paper provides a comparative analysis of transformation engines ranging from publicly or commercially available engines to be adapted to the mapping problem to hand-coded transformation engines that we developed for the sole purpose of supporting the mapping problem. The results of the comparison let authors to identify the type of transformation engine that fits better to their skills, needs and preferences.

1. Introduction

A User Interface (UI) model consists in a series of representations (called component models) pertaining to various facets of the UI such as: user’s task, domain objects, UI presentation and dialog, user’s characteristics, computer platform, physical environment interaction, etc [15, 23]. The UI specification is done with a set of transformation rules taken from a transformation catalog [15]. However, building a UI along with an explicit mechanism to manipulate the abstractions needed to support this process throughout a development cycle is still an open issue [9], referred as the mapping problem by Puerta and Eisenstein [19]. Considering a Multi-Path development of UIs as proposed in [15] the complexity of the problem is based on the increasing number of transformation needed to
support the process, illustrated in Fig. 1. Each development path (for instance, forward engineering) is composed of development steps, the latter being decomposed into development sub-steps (for instance, from abstract to concrete models). A development sub-step is realized by one (and only one) transformation system and a transformation system is realized by a set of graph transformation rules.

Fig. 1. Articulation of graph transformations with transformational development of UIs

In the last few years we have been working on the lack of disciplined and explicit mechanism for supporting UI development using a transformational approach [9, 14, 24]; and incorporating and manipulating design knowledge in user interface creation tools [24]. In addition, the User Interface eXtensible Markup Language (UsiXML) [15], a User Interface Description Language (UIDL), is grounded on our methodology. Our MDUID method relies on the Cameleon Reference Framework [3], a flexible approach towards model-driven UI design. This framework is composed of four development steps: create conceptual models (e.g. task model, data model, user model), create Abstract UI (AUI), create Concrete UI (CUI), and create Final UI (FUI). To detail the different techniques, to compare and assess them is beyond the scope of this paper, but we consider the set of transformations proposed above as the basis for the model transformations. For more details see [26]. The purpose of this paper is intended to show the lesson learned from using existing transformation engines, YATE, TransformiXML [14], AGG [7] and AToM³ [6], to support the transformational approach proposed in UsiXML. In addition, a new tool called YATE, developed from scratch in Java, is compared with the results generated by other tools. The remainder of this paper is structured as follows: Section 2 describes YATE implementation. Then, in Section 3 a case study is introduced to illustrate the different implementations in TransformiXML, AGG, YATE and AToM³. In Section 4, a comparative analysis is presented showing that this type of work remains unprecedented. The conclusion summarizes the main benefits of the evaluation, while contrasting with potential shortcomings.

2. Transformation engines for User Interface Development

Adopting a model-driven approach for the development of UIs requires a mechanism to support the mapping problem. Today, several research has been conducted addressing the problem, UsiXML [26], ATL [11], TXL [4], 4DML [2],
UIML's internal transformation capability [1], XSLT [12], GAC [8], RDL/TT [22], TEALLACH [10], TERESA [17], and UI Pilot [20]. A complete review, Table 1, of some of these tools has been described their characteristics has been introduced in [18, 21]. However, most of those tools support one-to-one mappings, which is a limitation in the context of multi-path development where the mappings are normally of the type one-to-many. When they support one-to-many mappings the implementation, sometimes is very complex and maintainability becomes even worse, we got this experience while using AToM³ [6].

Table 1. Comparison of general transformation language features (Adapted from [21]).

<table>
<thead>
<tr>
<th>Feature</th>
<th>ATL</th>
<th>GT</th>
<th>TXL</th>
<th>4DML</th>
<th>XSLT</th>
<th>GAC</th>
<th>UIML</th>
<th>RDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declarative</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Imperative</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Model Transformation</td>
<td>+</td>
<td>+</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
</tr>
<tr>
<td>XML Transformation</td>
<td>-</td>
<td>-</td>
<td>(+)</td>
<td>(+)</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Code Transformation</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>(+)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Complex Mapping</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Extensible</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Parameterizable</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
</tbody>
</table>

For these reasons, we build from scratch a new transformation tool YATE and compare its performance with other transformation engines that are similar in some extent; TransformiXML, AGG and AToM³ rely on graph grammars and graph transformations. Some other languages could have been also considered but we just limited our survey to those in which we have practical experience. A common case study was use to compare the tools, which is described in next section.

2.1 Case Study: Virtual Polling System

The virtual polling system case study applies a transformational approach in order to develop a UI aiming at collecting opinions of users. The development scenario is the following: a forward engineering path is applied from a definition of the task and domain viewpoint to produce both an abstract user interface (AUI) and concrete user interface (CUI). For that purpose more than one hundred rules are applied. The transformation rules and the details of this example are fully described for multi-modal applications in [24] and for Three-Dimensional UIs in [9]. For this case study around 60 transformations are applied, but, for simplicity, just four rules are detailed here to illustrate how they were implemented in AToM³, AGG, TransformiXML and YATE. The scenario describes transformations from task model to AUI model (Fig. 2).

The list of rules to perform the transformation is:
1. For each task that has task children an abstract container (AC) will be created. For instance the root task participate to opinion pool has three task children (provide personal data, answer question and send the questionnaire) so an abstract container (AC1) is created.

2. For each leaf task an abstract individual component (AIC) will be created. For instance for each leaf task, create name, create zip code, select sex, select age category, show questionnaire, select answer, send questionnaire an AIC will be created.

3. For each parent task that has children tasks, If parent task is associated to an AC (called parent AC) and child task is associated to an AC (called child AC), then, create an association relationship that will ensure the containment of the child AC into the parent AC. For instance the task participate to opinion pool associated to AC1 has two task children (provide personal data, answer question) that are respectively associated to AC11 and AC12. An association relationship is created to ensure the containment of AC11 and AC12 into AC1.

For each parent task that has children tasks, If parent task is associated to a parent AC and child task is associated to a child AIC, then, create an association relationship that will ensure the containment of the child AIC into the parent AC. For instance the task answer question is associated to AC12 has two children (show questionnaire, select answer) that are respectively associated to AIC121 and AIC122. An association relationship is created to ensure the containment of AIC121 and AIC122 into AC12.

![Fig. 2. Mapping between the task model and the abstract model.](image)

### 2.2 YATE Implementation

First, YATE a transformation engine developed in Java to support UIs development is introduced. The goal of the tool is to apply model-to-model transformations to support MDUID. The main class uses as input a UsiXML file and translates it into Java objects, and instantiates the rules class. Then, the rules are executed and the result is stored in the main class. Methods for applying the transformation rules are stored in the rules class and are complemented by the class
ruleshelper, which contains methods to find objects using iterators. Finally, a rulesTree class contains the tree of the transformation rules. The current version of the tool supports just a set of transformation rules for forward engineering. If any new rule is added this modifies the classes: rules with its corresponding helpers and the rules tree. The execution of transformation rules (Fig. 3) starts converting a UsiXML file into Java objects. Then, transformation rules are executed and for some transformation rules the RulesHelpers class is instantiated. Finally, the Java objects are re-converted into the UsiXML format.

Fig. 3. Sequence diagram of the application

The graphical user interface of this hand coded transformation engine (Fig. 4) was developed using Java Swing. It is composed of a very simple UI with big text areas used to show the source (1) and target files (2); the tree containing all the implemented transformation rules (3) each with a checkbox aside allowing the decision whether or not the rule will be executed. The main class uses JAXB to internally represent with Java objects the UsiXML file. The Castor project is used to map a file containing two inputs: the specification of UsiXML and the transformation rules described also is a XML format. The rules class defines all the methods for the transformation rules. As a consequence, there is a loss in performance coming from the fact that different rules can apply on the same object and each one will have to do the search to find this object. A complete description of the rules can be accessed in [5].

A couple of rules are described in this section to illustrate their implementation. First, the code to map files corresponding to Abstract Container objects is as follows: the class usixml.po.AbstractContainer, see code above, is mapped in the new XML file with the name abstractContainer. Then all attributes of the class AbstractContainer are mapped into the new XML file, for each attribute the way it formatted (attribute or node) is specified. The last attribute is a list, with bind-xml=element and auto-naming=deriveByClass, which means that the name of the attribute (in the new xml file) will be dynamically chosen by Castor, following the name of the class. For instance, an AioType can be an AbstractContainer, an Ab-
strictIndividualComponent or an Input (all extend AioType), so the name of the attribute can be one of these three. There is one method per transformation rule. Consequently, the performance of the tool is affected due to constant use of objects to apply one rule at a time. Even though, maintainability of the code is possible following this approach. The main difficulty encountered was the unavailability of a dedicated pattern matching API, therefore, searching a LHS in the UsiXML file and replacing it with the RHS is not possible. The code shown hereafter corresponds to the transformation rule defining an abstract adjacency. The rule express the adjacency between sibling tasks executed in AIOs, as they are siblings we can infer that they must be next to each other in the Final User Interface.

Fig. 4. Graphical User Interface of UsiXML Transformation Engine.

2.3 AToM³ Implementation

The structure of UsiXML is represented as an oriented graph in AToM³ [6]. The implementation in AToM³ of UsiXML meta-models, see Fig. 6, presents some shortcomings, for instance the aggregation relationship expressing an AC is composed of AIC became two relationships: an AC contains AIC and AIC are contained in AC. As UsiXML has around 60 aggregation relationships, this limitation increases the number of connecting arrows on the diagram just for this type of relationship to 120.

In Fig. 5 (1) correspond to tasks, (2) to ACs and (3) to AICs. The left hand side (LHS) of the transformation engine corresponds to the condition to search on the graph and the right hand side (RHS) part corresponds to the transformation re-
sult. Avoiding an infinite search cycle of the condition is done by the specification of preconditions, coded in Python. At least two actions must be performed to apply a rule. First, set a post-condition to the nodes visited and mark them as visited. Second, define a precondition to check whether the node has been visited or not before applying the transformation. The designer must keep in mind these conditions and code them when necessary.

Fig. 5. Rules transformation expressed in AToM³

Fig. 6. Task and abstract user interface meta-models expressed in AToM³

2.4 AGG Implementation

AGG [7] provides a GUI enabling the specification of the transformation rules and a customizable interpreter enabling their application through the API.
One of the advantages that AGG provides is its capacity to graphically specify the negative application conditions (NAC) or preconditions, LHS and RHS. For the case study the rules (Fig. 7) were more easily encoded using the NAC rule editor. AGG provides: (1) a programming language enabling the specification of graph grammars and (2) a customizable interpreter enabling graph transformations.

AGG allows the graphical expression of directed, typed and attributed graphs (for expressing specifications and rules). It has a powerful library containing notably algorithms for graph transformation, critical pair analysis, consistency checking, positive and negative application condition enforcement. Experiments showed that AGG is a rigorous environment for defining and applying rules. Unfortunately, it shows poor in terms of usability for specifying large UI models. Indeed, it may appear somewhat abstract to the designer to describe a UI appearance with a set of nodes and relationships. The main components are summed up as follows:

1. **A design editor** allows the creation and the consolidation of models exploited in the development process. A specific environment enables the design of UI appearance by direct manipulation of widgets.
2. **A design derivator** enables model to model transformations.
3. **A rule editor** enables the definition of new transformation rules.
4. **A rule validator** enables the designer to identify conflicts within a set of rules. The critical pair analysis technique is used for this purpose.
5. **A design analyzer** enables the verification of desirable properties of the manipulated artifacts such as basic consistency rules, type checking or even usability properties (i.e., IFIP properties like reachability, browsability).

Fig. 7. Rules transformation expressed in AGG.
2.5 TransformiXML Implementation

TransformiXML [13] is an environment that addresses the mapping problem (any type of structural mapping) by supporting a mathematical expression of the relationships (based on graph grammars) and allowing the definition and the application of transformation rules. This environment is sub-divided into two components: an Application Programming Interface (TransformiXML API) that can be used by any application to apply transformation rules in a batch-like way (non-interactive) and a Graphical User Interface that serves as a front-end application to the API (TransformiXML GUI) in an interactive way. The second component can drive a transformation process involving several models in a UI development process. Both components share a generic requirement: to manipulate any UI model expressed in UsiXML and to apply transformation rules over them. TransformiXML API relies on AGG API, consequently, transformation rules are expressed exactly as in AGG (Fig. 7). TransformiXML uses a catalog of predefined transformation rules, allows the execution of transformations in an interactive manner, and finally allows the association of development sub-steps with transformation systems. The basic flow of tasks with TransformiXML GUI is the following: users choose an input file containing models to transform. Then, they choose a development path by selecting a starting point and a destination point (e.g., the viewpoint to obtain at the end of the process). Depending on the content of the input file some of the development paths may not be available. A tree allows users to visualize the proposed development model (i.e., all the steps and sub-steps for a chosen path). Users can load another development model for the selected path. Now the task of the user consists in attaching one transformation system for each development sub-step. By clicking on a sub-step in the tree, a set of transformation systems realizing the chosen sub-step are displayed. A transformation system may be attached to the current sub-step by clicking “Attach to current sub-step”. The user may also want to edit the rules either in an XML editor or in AGG environment. After attaching a transformation system for each rule in the development model, the user may apply the transformation either step by step or as a whole. The result of the transformation is then explicitly saved in a UsiXML file.

3. Comparative Analysis of Transformation Engines

In this section a comparative analysis to transformation engines is presented. This information can be relevant to HCI community while trying to tackle the mapping problem for UI development. The selected criteria, summarized in Table 2, were chosen considering the key factors that are relevant for: (1) implementing of a transformation engine (implementation paradigm and required programming skills, code generation support, pattern matching API), (2) usability (rules organization, rules scheduling organization) and further use of the transformation engines (maintainability and flexibility and completeness). The description of each criterion is as follows:
• Implementation paradigm and required programming skills. First major difference between the transformation engines is whether they are imperative or declarative. Like all graph transformation tools, AToM³ and AGG are strictly declarative while YATE and TransformiXML are imperative. Performance is a very important criteria and AToM³ posses a major disadvantage on this aspect. As AToM³ is entirely coded in Python and compiled at execution the execution is slower compared to precompiled Java code, YATE, AGG and TransformiXML. Moreover, the fact that the transformations are graphically designed means that the user does not have the possibility to optimize how they are executed, the engine is in charge of that. YATE and TransformiXML are faster during execution. Not only because the compiled code is faster that the interpreted code of AToM³ but also the programmer chooses exactly how the transformation will be executed. Redundancy can be eliminated as well by grouping transformation rules. While AToM³ needs programming skills, as there are a few things to code in Python, like conditions, actions, constraints and variables’ valuing, AGG requires programming skills to specify conditions on attributes. However most of the transformation are described graphically in AGG and AToM³, consequently they are more flexible because even GUI designers with (almost) no programming skills can use them, and also because maintaining the rules is simpler. Nevertheless, in AGG and AToM³, the transformation rules are specified for the meta-model that we have graphically created and a change to this meta-model will make the transformations not work anymore.

<table>
<thead>
<tr>
<th></th>
<th>AToM³</th>
<th>TransformiXML</th>
<th>YATE</th>
<th>AGG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation paradigm</td>
<td>Declarative</td>
<td>Declarative</td>
<td>Imperative</td>
<td>Declarative</td>
</tr>
<tr>
<td>Required programming skills</td>
<td>Medium, short learning period</td>
<td>None</td>
<td>High, long learning period</td>
<td>Medium</td>
</tr>
<tr>
<td>The model-to-model approach</td>
<td>Graph transformation</td>
<td>Graph transformation</td>
<td>Hybrid</td>
<td>Graph transformation</td>
</tr>
<tr>
<td>Code generation</td>
<td>Not Supported</td>
<td>Supported</td>
<td>Supported via external tools</td>
<td>Not supported</td>
</tr>
<tr>
<td>Pattern matching</td>
<td>Supported</td>
<td>Supported</td>
<td>Supported via external tools</td>
<td>Supported</td>
</tr>
<tr>
<td>Rules scheduling organization</td>
<td>Fixed order, no dynamic flow control</td>
<td>Explicit flow control</td>
<td>Explicit flow control</td>
<td>Fixed order, no dynamic flow control</td>
</tr>
<tr>
<td>Rules organization</td>
<td>Distinct rules sets, No limitations no inheritance</td>
<td>No limitations</td>
<td>Distinct rules sets, no inheritance</td>
<td></td>
</tr>
<tr>
<td>Flexibility</td>
<td>Very good</td>
<td>Good</td>
<td>Very poor</td>
<td>Very good</td>
</tr>
<tr>
<td>Maintainability</td>
<td>Good</td>
<td>Very poor</td>
<td>Very poor</td>
<td>Good</td>
</tr>
<tr>
<td>Completeness</td>
<td>Average</td>
<td>Very Good</td>
<td>Very Good</td>
<td>Average</td>
</tr>
</tbody>
</table>

Table 2. Comparison of transformation engines.

• The model-to-model approach. TransformiXML, AGG and AToM³ use graph
transformations. The models are created graphically and so are the transformation rules. This is a very intuitive and easy to read and maintain. YATE uses another approach. Instead of transforming the model into a graph, it directly modifies it. In fact, the XML file is read and transformed into Java objects. These objects are then modified by the transformation rules and finally the Java objects are translated back into XML. In YATE, a method is created for each transformation rule. Comparing with the graphs the code is harder to read and maintain.

- **Code Generation.** In YATE, code generation is feasible; the Castor API supports this process. On the contrary, AGG and AToM³ code generation is possible but it is poorly developed. To generate XML code for a model in AToM³, we should use transformation rules, with the “action” code writing XML code to a text file. This would be long and fastidious, if possible.

- **Pattern matching.** This is a big difference among all the tools. While pattern matching is supported by TransformiXML, AGG and AToM³, YATE does not support it for the moment. Pattern matching is still usable in YATE, but with the help of external projects or at the price of a long a fastidious implementation. This is more complex to implement than in TransformiXML, AGG and AToM³.

- **Rules scheduling and organization.** AGG and AToM³ only allow determining a fixed order on the rules. There is no possibility of explicit flow control on them. And there is not a “call” instruction for a rule to call another. In fact, pre-conditions serve to decide whether or not the rule will be executed. In AToM³ it is possible to create distinct set of rules and execute one after another. However, there is no rule inheritance mechanism and import mechanism either. Therefore rule sets can not import a rule from another set and a rule can neither use another. Finally, TransformiXML and YATE, allow rules organization (for example one class for each rules set), and the flow control is of course explicit. In YATE it is possible even to call a rule in another, because each rule is a method in Java.

- **Flexibility and maintainability.** The more readable the models the easiest to maintain them. AGG, AToM³ and TransformiXML offer a best capacity to modify and maintain meta-models, models and transformation rules. However, when modifying rules at least two issues must be taken into account: 1) if a meta-model is modified then transformation rules designed for that meta-model are potentially not working anymore (because they can apply on objects removed or modified); 2) modifying rules is sometimes difficult, for instance in AToM³ modifications on rules leads to unexpected behavior. Finally, because of its fully programmatic approach and more complex syntax, YATE is the most difficult to maintain and modify, as it implies several modification in several classes in addition to coding the transformation rules. So, flexibility is worse than for the two other three tools.

- **Completeness** refers to the ability of the system to handle complex rules and generate code. As AToM³ allows programming in Python, it is able of execut-
ing more complex rules than strict graph-transformation rules (with only NAC, LHS and RHS). Still, the lack of explicit flow control and the absence of imperative constructs limit it. Because the source and target model are not distinct, they both obviously can be navigated and modified. Finally, the lack of code generation makes AToM³ less complete. Finally, in YATE the limit is in fact the programming skills of the graphical interface designer.

Finally, Fig. 8 shows variations between completeness/performance and maintainability/flexibility. While it is known that robust and complete software is desired, this is difficult to achieve. HCI is a discipline particular with constant changes so it can be said that incompleteness is intrinsic to HCI. However completeness on existing needs is still relevant and for UI development it seems that TransformiXML and YATE are more complete than AGG and ATOM³. On the other hand, the lack of maintainability and flexibility contrast with the other tools. Authors have to choose the equilibrium that fits better their particular needs.

4. Conclusion

In this paper, we introduced a comparison of transformation engines to support UI development. Four tools were compared, including two developed in our lab, using a common case study. From our evaluation we noticed that it is difficult to find the perfect balance with the criteria. In particular, one of the goals of model-driven development of UIs is to support UI run-time adaptation based on transformation rules. Therefore, the performance is of concern. We conclude our report with a comparative analysis that could help authors to make the decision of the appropriate transformation engine to be used while confronting model-driven engineering of UIs. In this area, we can conclude that 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 for UIs:
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. 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.
3. 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.
4. **Acceptance and adoption** of this model-driven approach requires industry standards to provide openness to consumers, and foster competition among vendors

Not all model-based UI development environments or development methods can pretend to be compliant with these principles. If we apply OMG’s principles to the UI development life cycle, it means that models should be obtained during steps of development until 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. The following definition was approved unanimously by 17 participants of the ORMSC plenary session meeting in Montreal on 23-26 August 2004. The stated purpose of these two paragraphs was to provide principles to be followed in the revision of the MDA guide.

MDA is an OMG initiative that proposes to define a set of non-proprietary standards that will specify interoperable technologies with which to realize model-driven development with automated transformations. Not all of these technologies will directly concern the transformation involved in MDA. MDA does not necessarily rely on the UML, but, as a specialized kind of MDD (Model Driven Development), MDA necessarily involves the use of model(s) in development, which entails that at least one modeling language must be used. Any modeling language used in MDA must be described in terms of the MOF language to enable the metadata to be understood in a standard manner, which is a precondition for any activity to perform automated transformation.

**Acknowledgments** We gratefully thank the support from the SIMILAR network of excellence, supported by the 6th Framework Program of the European Commission, under contract FP6-IST1-2003-507609 (http://www.similar.cc), the Alban program (www.programalban.org) supported by European Commission, and the CONACYT (www.conacyt.mx) program supported by the Mexican government. All information regarding UsiXML is accessible through http://www.usixml.org.

**References**
17. Mori G., Paternò F., Santoro C. Design and development of multidevice user...


