A Comparative Analysis of Graph Transformation Engines for User Interface Development

Juan Manuel González Calleros[1], Adrian Stanciulescu[1], Jean Vanderdonckt[1], Jean-Pierre Delacre[1], Marco Winckler[1,2]

[1]Université catholique de Louvain, Louvain School of Management (LSM)
Place des Doyens, 1 – B-1348 Louvain-la-Neuve (Belgium).
E-mail: {juan.gonzalez@student, jean.vanderdonckt@, adrian.stanciulescu@}uclouvain.be
Web: http://www.isys.ucl.ac.be/bchi/ - Tel: +32 1047 {8349, 8525} – Fax: +32 10 478324
[2]LIIHS-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. User Interface mappings are also relevant to web engineering. We have been working not just User Interface mappings for web-based systems but as well for Information Systems in general. However, we have been confronted to the mapping problem as the use of an appropriate transformation tool it still an issue in our research group. Although several transformation engines support mappings a transformation engine capable of supporting a transformational approach for ensuring model-driven engineering of user interfaces is still an open issue. 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

Model engineering (i.e., a discipline that is concerned with the development of models) is part of the numerous solutions proposed to overcome with the increasing complexity that developers must handle to produce software. Model-driven development (MDD) is an OMG (www.omg.org) 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 transforma-

tions. It advocates that software development should be guided as much as possible by the construction, and refinement of software models at various levels of abstraction. Four principles underlie the OMG's view of MDD for User Interfaces (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 soft-ware.
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 [24]. 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. MDD has been applied to many kinds of business problems and integrated with a wide array of other common computing technologies. Considering MDD of UIs [15] complexity related to the number of transformation needed to support this process has been found in the literature as a major issue [13]; In Fig. 1 how graphs transformations area articulated when following MDD method. 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.

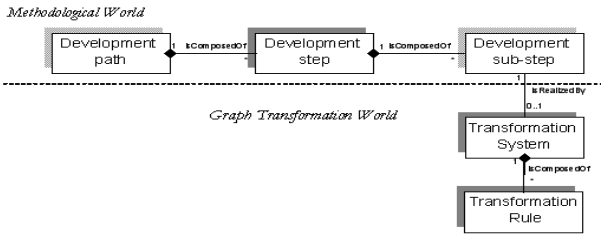

Fig. 1. Articulation of graph transformations with transformational development of UIs

The amount of transformation rules of a rather simple system grows up to two hundred graph transformation rules. Then, usable systems are needed to encode

transformations rules. Also, provide facilities to extend existing sets of graph transformations. Recently, we have been working on this issue. On the one hand, trying to use existing tools to encode graph transformation rules applied to MDD of UIs. On the other hand, building from scratch customized tools [14, 23] addressing the same problem.

In this paper the lesson learned from using graph transformation tools is discussed. Graph Transformation engines were compared and their benefits and constraints are presented to authors with the goal of helping in making decisions when confronted to selecting a graph transformation tool for MDD of UIs. Our analysis is based on three existing tools TransformiXML [14], AGG [7] and AToM³ [6]. A new tool called YATE (Yet Another Transformation Engine) recently developed for MDD of UIs is also compared with existing ones. 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

Several research has been conducted addressing the mapping problem for supporting MDD of UIs, UsiXML [24], 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]. Some of these tools were compared in [18, 21]. The results shows that most of existing solutions support one-to-one mappings, which is a limitation in the context of MMD of UIs where the mappings are normally, not always, of the type one-to-many. When a tool support one-to-many mappings implementation, sometimes, is very complex and other issues raised, such as: maintainability and usability.

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

Feature	ATL	GT	TXL	4DML	XSLT	GAC	UIML	RDL
Declarative	+	+	+	+	+	-	+	-
Imperative	+	-	-	-	+	+	-	+
Model Transformation	+	+	(+)	(+)	(+)	(+)	(+)	(+)
XML Transformation	-	-	(+)	(+)	+	+	-	+
Code Transformation	-	-	+	(+)	-	-	-	-
Complex Mapping	+	+	+	+	+	+	-	+
Extensible	+	-	-	-	-	-	-	+
Parameterizable	-	-	-	-	-	+	-	+

In Table 1 if a feature is supported, it is marked wit a “+”, if not it is marked

with "-". If a supported feature is put in brackets, it means that it is in principle supported (maybe with some additional effort) but that the language is not specifically designed to support that property. From this review [21] they conclude that the effective tool depends on the approach selected. It was found that for purely MDD, graph transformations and ATL will be good choices [21]. From our experience, we found graph transformations a good option for MDD of UIs. The selection of graphs transformation [13,14,15] were based on the fact that graph grammars:

- are rather declarative and provide an appealing graphical syntax which does not exclude the use of a textual one
- are based on a formally defined execution semantics based notably on pushout theory, for which many proofs have been provided (completeness; confluence)
- allow to describe transformations with the same vocabulary as specification models in a very consistent manner and for all development steps
- provide extensions (i.e., conditional graph rewriting, typed graph rewriting) to check important properties of the artifact that is produced after a transformation
- offer modularity by allowing the fragmentation of complex transformation heuristics into small, independent chunks. The fact that graph rewritings have no-side effects facilitates this modularization.

2.1 Case Study: Virtual Polling System

Our method UsiXML [24] relies on the Cameleon Reference Framework [3]. The 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 for more details see [24]. In this section we use the virtual polling system case study to exemplify the transformational approach based on graph transformations.

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 [23] and for Three-Dimensional UIs in [9]. For this case study around 6O 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.
4. 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 AIC 122). An association relationship is created to ensure the containment of AIC121 and AIC122 into AC12.

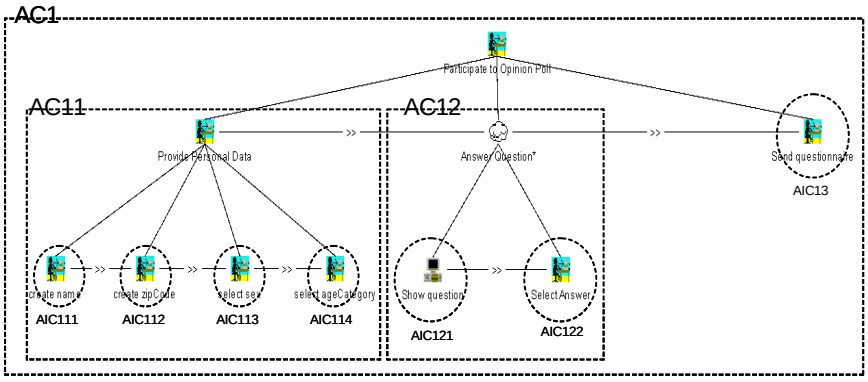

Fig. 2. Mapping between the task model and the abstract model.

2.2 YATE Implementation

In this paper, the aim is to compare several transformation tools with respect to a selected set of criteria considered of importance for MDD UIs. For this reason we included the programming model, since it may be a premier choice for a developer being familiar with either declarative or imperative programming [21]. As already described a common case study is used to illustrate its implementation on the different tools. YATE transformation engine was developed in Java. In practice, see sequence diagram in Fig. 3, the *main class* uses as a UsiXML file (which is xml) and translates it into Java objects, then instantiates the *rules* class. 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. After, the transformation rules are executed and the result is stored in the *main* class. Graph transformations are stored as methods in the *rules class.* The *class ruleshelper* 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 small set of transformation rules for forward engineering for the polling system. If any new rules are needed then a lot of modifications are needed. The *rules* class, its corresponding *helpers* and the *rules tree* class need and update. Then maintainability of the code became an issue as the number of transformation rules increase. Due to the fact that *rules* class defines all the methods for the transformation rules. As a consequence, there is a lost 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].

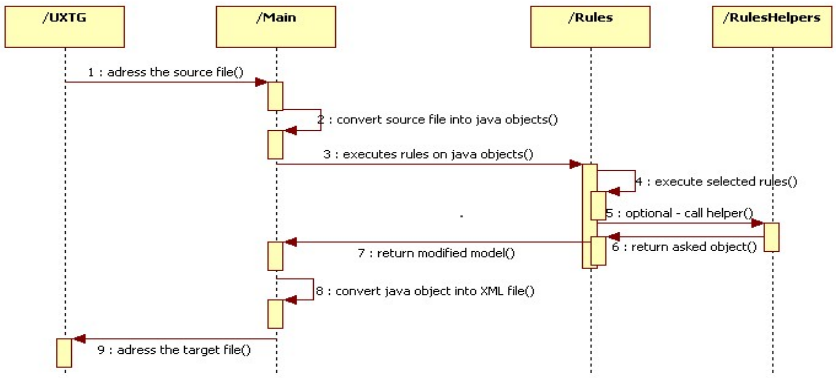

Fig. 3. Sequence diagram of the application

The graphical user interface of YATE (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 executeThe code to map files corresponding to *Abstract Container objects* is as follows: the class *usixml.po.AbstractContainer* 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 *AbstractIndividualComponent* 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 ob-

jects to apply one rule at a time. Even thought, 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.

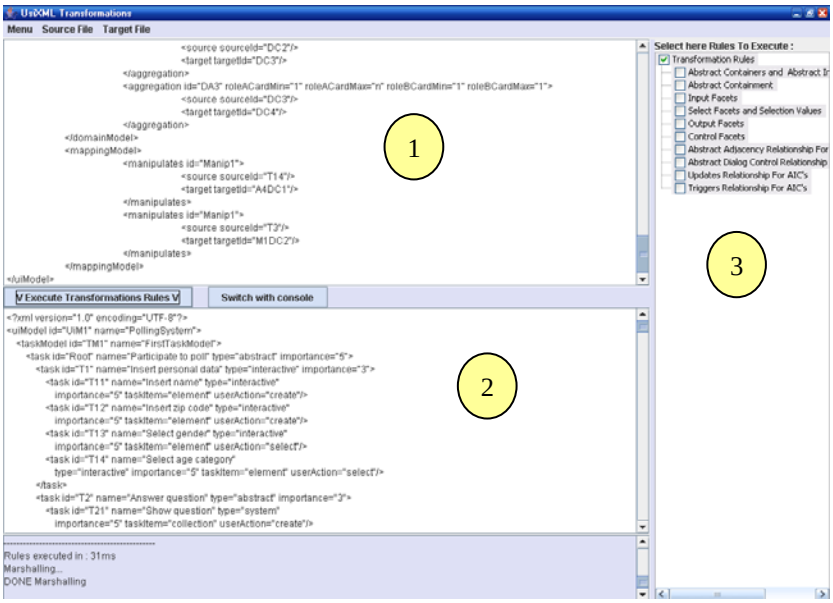

Fig. 4. Graphical User Interface of UsiXML Transformation Engine.

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 result. Avoiding an infinite search cycle of the condition is done by the specification of preconditions, coded in Python. At least two actions most 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 mine these conditions and code them when necessary.

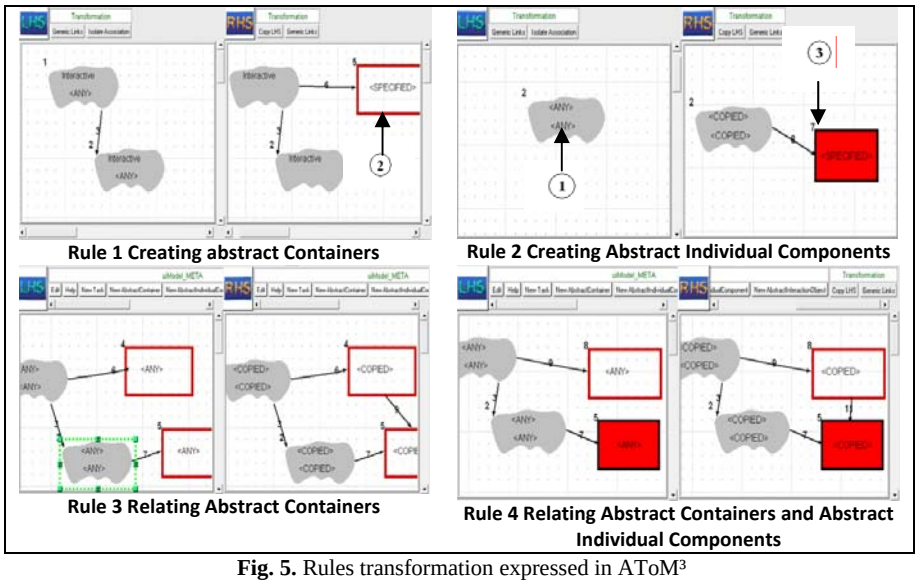

Fig. 5. Rules transformation expressed in AToM³

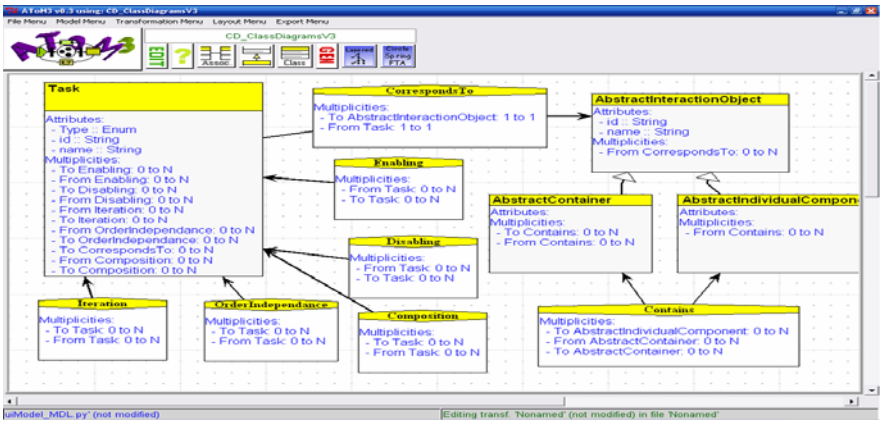

Fig. 6. Task and abstract user interface meta-models expressed in AToM³

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).

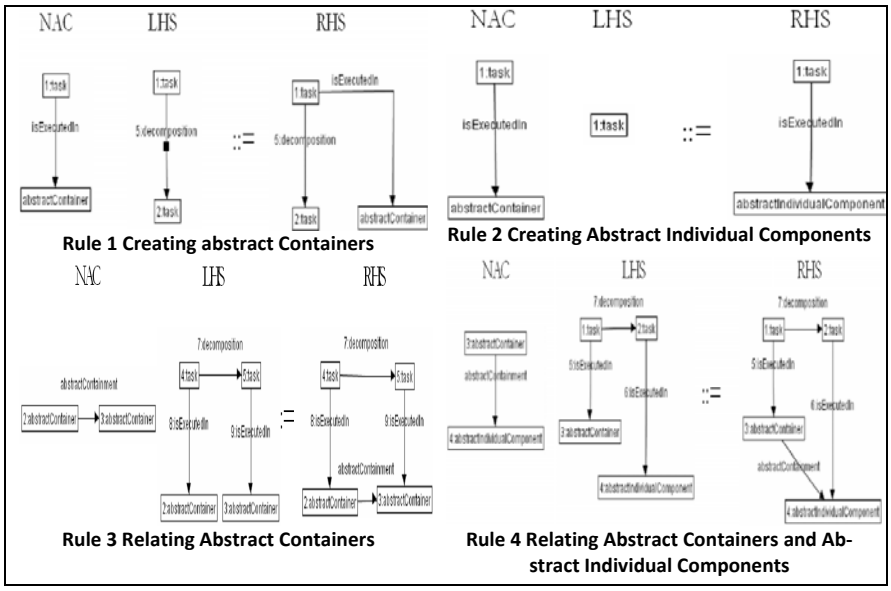

Fig. 7. Rules transformation expressed in AGG.

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 de-*

clarative. 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 fasters 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 2. Comparison of transformation engines.

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

- *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, preconditions 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 ATOM^3. 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.

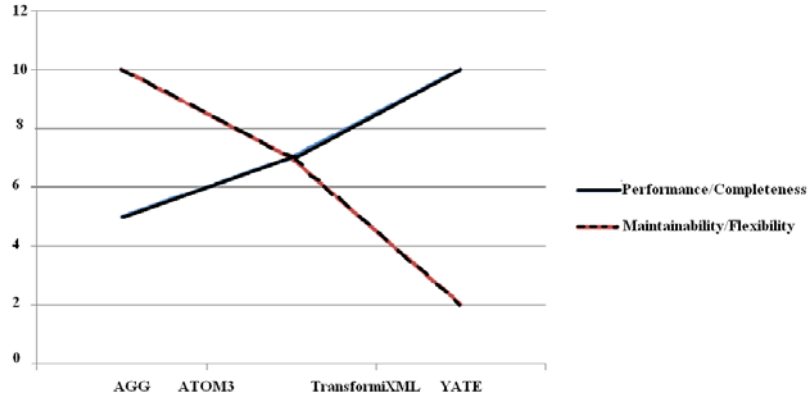

Fig. 8. Tools comparison with respect to performance and maintainability.

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. 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

1. Abrams, M. and Helms, J., User Interface Markup Language (UIML) Specification, Working Draft 3.1. OASIS, 2004.
2. Brown, S.S., Conversion of notations. Tech. Rep., Univ. of Cambridge, 2004.
3. 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.
4. Cordy, J.R., the TXL source transformation language. Science of Computer Programming, 61:190–210, August 2006.
5. Delacre, J.-P., A Comparative Analysis of Transformation Engines for User Interface Development, M.Sc. thesis, UCL, Louvain-la-Neuve, 28 August 2007. http://www.isys.ucl.ac.be/bchi/publications/2007/Delacre-MSc2007.pdf
6. de Lara, J., Vangheluwe, H., AToM³: A tool for multi-formalism and metamodelling, in: FASE '02: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering (2002), pp. 174-188.
7. Ermel, C., Rudolf, M. and Taentzer, G., The AGG-Approach: Language and Tool Environment. H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg (eds.). Handbook on Graph Grammars and Computing by Graph Transformation. Vol. 2: App., Languages and Tools. pp. 551–603. *World Scientific*, 1999.
8. Fiala, Z. and Houben, G.-J., A generic transcoding tool for making web applications adaptive. In Proc. of the CAiSE'05 Forum. CEUR Workshop Proceedings, online CEUR-WS.org/Vol-161/FORUM 03.pdf, 2005.
9. Gonzalez, J.M., Vanderdonckt, J., Arteaga, J.M., A Method for Developing 3D User Interfaces of Information Systems, Proc. of CADUI'2006 (Bucharest, 6-8 June 2006), Chapter 7, Springer-Verlag, Berlin, 2006, pp. 85-100.
10. Griffiths T., Barclay, P.J., Paton, N.W., McKirdy, J., Kennedy, J.B., Gray, P.D., Cooper, R., Goble, C.A., da Silva, P.P. Teallach: A Model-based user interface development environment for object databases. *Interacting with Computers 14*, 1 (2001), 31–68.
11. Jouault, F. and Kurtev, I., Transforming models with ATL. In Proc. of MoDELS 2005 Workshops, LNCS, Vol. 3844, pp. 128–138. Springer, 2006.

12. Kay, M., XSL Transformations (XSLT) Version 2.0, W3C Working Draft. W3C, April 2002.
13. Limbourg, Q., Vanderdonckt, J., Addressing the Mapping Problem in User Interface Design with UsiXML, Proc. of 3rd Int. Workshop on Task Models and Diagrams for user interface design TAMODIA'2004, 2004, pp. 155-163.
14. Limbourg, Q. and Vanderdonckt, J., Transformational Development of User Interface with Graph Transformations, Proc. of CADUI 04 (Funchal, 12-14 January 2004), *Kluwer Academics*, Dordrecht, 2005.
15. Limbourg, Q., Vanderdonckt, J.: UsiXML: A User Interface Description Language Sup-Porting Multiple Levels of Independence. In: Matera, M., Comai, S. (eds.): Engineering Advanced Web Applications. Rinton Press, Paramus (2004), 325–338.
16. Paternò, F., Santoro, C. A unified method for designing interactive systems adaptable to mobile and stationary platforms. *Interacting with Computers 15*, 3 (2003) 349–366.
17. Mori G., Paternò F., Santoro C. Design and development of multidevice user interfaces through multiple logical descriptions. *IEEE Transactions on Software Engineering 30*, 8 (2004), 507–520.
18. Pérez-Medina, J. L., Dupuy-Chessa, and Front, A., A Survey of Model Driven Engineering Tools for User Interface Design. In 6th International Workshop on TAsk Models and DIAgrams TAMODIA'2007, Toulouse, France, Novembre 2007), LNCS 4849.
19. Puerta, A. and Eisenstein, J., Towards a General Computational Framework for Model-Based Interface Development Systems Model-Based Interfaces, in Proc. of ACM International Conference on Intelligent User Interfaces IUI'99 (Los Angeles, 5–8 January 1999), *ACM Press*, New York, 1999, pp. 171–178.
20. Puerta, A., Micheletti, M., and Mak, A. The UI Pilot: A Model-Based Tool to Guide Early Interface Design. In Proc. of IUI'2005, *ACM Press*, New York, (2005), 215–222.
21. Schaefer, R., A Survey on Transformation Tools for Model Based User Interface Development, Proc. of 12th Int. Conf. on Human-Computer Interaction HCI International'2007 (Beijing, 22-27 July 2007), Part I, Lecture Notes in Computer Science, Vol. 4550, Springer-Verlag, Berlin, 2007, pp. 1178-1187.
22. Schaefer, R., Mueller, W., Dangberg, A., RDL/TT – a description language for the profile-dependent transcoding of xml documents. In Proceedings of the first International ITEA Workshop on Virtual Home Environments, 2002.
23. Stanciulescu, A., Limbourg, Q. Vanderdonckt, J., Michotte, B. and Montero, F.: A transformational approach for multimodal web user interfaces based on UsiXML. Proc. of 7th Int. Conf. on Multimodal Interfaces ICMI'2005 (Trento, 4-6 October, 2005), ACM Press, New York, 2005, pp. 259-266.
24. Vanderdonckt, J. A MDA-Compliant Environment for Developing User Interfaces of Information Systems. In *Proc. of CAiSE'05*, Springer-Verlag, Berlin (2005), 16–31.