A Proposal for Enhancing the UsiXML Transformation Meta-Model

Nathalie Aquino, José Ignacio Panach, Oscar Pastor
Centro de Investigación en Métodos de Producción de Software
Universidad Politécnica de Valencia
Camino de Vera s/n, 46022 Valencia, Spain
+34 96 387 70 07 Ext. 83534
{naquino, jpanach, opastor}@pros.upv.es

ABSTRACT
UsiXML proposes a set of conceptual models that represent diverse aspects of user interfaces. Some of these models represent a user interface at a high abstraction level. From these high level models, other models at a lower abstraction level can be obtained by means of model-to-model transformations. Model-to-model transformations can occur between several abstraction levels. Finally, the user interface code is reached through a model-to-code transformation. Therefore, transformations are a fundamental piece in the UsiXML development process. Current UsiXML model-to-model transformation rules are specified using a graph-based notation. This strong dependency on graphs can result in a lack of efficiency in transformations when the amount of involved graphs is medium or large. In order to face this problem, we propose a transformation meta-model independent of graphs and any other transformation technology. The analyst can use this transformation meta-model to specify how transformations should be carried out throughout different abstraction levels. Once this specification has been finished, the analyst will be able to translate the transformation rules into a specific transformation language that performs the transformation. In the paper, we introduce an example to illustrate that a same transformation rule specified in our proposal can be translated into two different translation languages such as graph notation and ATL, depending on analyst's preferences.

Keywords
Model-driven engineering, UsiXML, meta-model, transformation, interaction modelling, conceptual model.

INTRODUCTION
Model-driven engineering (MDE) is a software development methodology which focuses on creating models, or abstractions, closer to concepts of a particular domain (i.e., at the problem space level) than to computing concepts (i.e., at the solution space level). According to [12], MDE is simply the notion that we can construct a model of a system that we can later transform into the real thing. MDE states that the analyst's effort must be gathered in a conceptual model, and the system is implemented later by means of transformation rules that can be automated or semi-automated. In other words, the MDE paradigm distinguishes between conceptual models, defined by analysts, and the code that implements the system, which can be generated with as much automation as possible from the corresponding conceptual model. In this context, model-to-model and model-to-code transformations are an essential piece to develop systems according to the MDE paradigm.

UsiXML [11] proposes an MDE method for user interface development. It comprises various types of high level models that allow to represent interactive tasks in a way that is independent of platforms and interaction modalities (e.g., character, graphical, vocal), as well as other lower level models that add the relevant details about platforms, devices, modalities, users, etc. Furthermore, UsiXML uses a transformational approach: higher level models are transformed to lower level models by means of model-to-model transformations, and the final user interface code is reached from lower level models by means of model-to-code transformations. Model-to-model and model-to-code transformations are specified by means of transformation rules.

The current proposal to perform model-to-model transformations in UsiXML is based on graph transformations [11]. The current version of the meta-model that characterizes this proposal is defined in the UsiXML V1.8 Reference Manual [21]. This meta-model incorporates classes which are too much related to the structure of graphs. This can be seen as a disadvantage, since it can result in a very high correlation between UsiXML and graphs transformations with their related tools. Furthermore, graph transformations have some disadvantages [22] that are inherited by UsiXML. For example, the number of nodes is critical. Graphs with many

nodes require much time to perform transformations. Another critical factor is the maximum degree of nodes. The more degrees, the more time is needed to carry out transformations. These disadvantages suggest that graph transformations would not be suitable for models of large size.

In the literature, there are other proposals related to transformations which are not focused on graphs, like the Query/View/Transformation (QVT) standard [18] and the ATLAS Transformation Language (ATL) [2]. QVT is the standard of the Object Management Group for specifying model transformations. It aims to transform MOF-based models. MOF (Meta-Object Facility) [14] is also a standard of the Object Management Group and is used for specifying meta-models. Its current version is MOF 2.0. The UML 2 meta-model (UML 2.0 and above) has been defined according to MOF 2.0, so all models compliant with UML 2 are MOF-based models and hence, can be transformed using QVT. QVT has been implemented in various languages. Among those languages, ATL is currently the most widely used [17]. ATL allows describing transformations using a syntax that is similar to the one of declarative and imperative programming languages. Since all UsiXML meta-models are defined using UML 2, we consider that features of languages that implement QVT, as ATL, are good options to take into consideration to perform UsiXML transformations.

In this paper, we propose some changes to the UsiXML transformation meta-model in order to enable the creation of transformation models in which rules are expressed in a way that is independent not only from the graph notation, but also from the specific syntax and notation of any transformation language. We incorporate concepts that are general enough to express the current UsiXML transformation rules that are expressed with the graph notation, as well as general transformation rules that can be expressed with languages like ATL. In fact, the proposed enhancements were inspired by features of ATL, taking care of not adopting its specific syntax and notation. Moreover, our proposed meta-model is also abstract enough to be used in other MDE methods that build user interfaces or fully functional software applications, like the OO-Method [15].

The main advantage of our proposal is that the analyst can specify transformations in a way that is abstract and independent of any specific transformation language. In a next step, it will be possible to automatically translate the transformation rules specified according to our proposed meta-model, to a selected transformation language. This translation must be performed according to the syntax of the target transformation language, such a way, transformations rules will be executed in the corresponding transformation technology. This transformation technology and language could be graph notation, ATL or other.

The remainder of the paper is structured as follows: Section 2 presents related works. Section 3 provides a brief description of UsiXML and its transformational approach. Section 4 presents our proposal for enhancing the current UsiXML transformation meta-model. Section 5 illustrates our proposal with an example. And, finally, Section 6 presents conclusions and future works.

RELATED WORK

In the literature there are many proposals to perform transformations. Graph grammars and graph transformations have been recognized as powerful techniques for specifying complex transformations that can be used in various situations in a software development process [1].

Besides graph transformations used in UsiXML, there are many other proposals based on graphs. For instance, Karsai [8] has defined a graph-transformation-based technique for specifying model transformations. Karsai proposes starting from a domain specific modelling language specified with UML class diagrams. Domain specific models are networks of objects, where each object (link) belongs to a corresponding class (association) in the class diagram. From a mathematical point of view, domain specific models are graphs where the labels denote the corresponding entities in the meta-model. According to the author, the design transformation process specified in this way is formal, and it assigns a semantic to the input models in terms of the target domain.

Another author who proposes performing transformations by means of graphs is Gogolla [6]. This author has studied transformation rules on the UML meta-model layer. The aim of those rules is to identify the minimal set of UML concepts, a UML core, which is necessary in order to express all UML language features. Due to the diagrammatical nature of UML, graph transformation is a natural language for the formulation of such rules. Specifically, Gogolla has defined transformations for class diagrams and statechart diagrams.

Other authors propose performing transformations by means of algebraic compositions. In this group, we can comment the work of Ho [7]. Ho has specified a freely available UML transformation framework for manipulating UML models, called UMLAUT. These manipulations are expressed as algebraic compositions of reified elementary transformations. Thus, they are open to extension through inheritance and aggregation. By means of UMLAUT, a UML model of a distributed application can be automatically transformed into a labelled transition system validated using a pre-existing protocol validation tool.

Other authors have defined more formal transformations than algebraic compositions. In this group, we would like to mention the work of Caplat [4]. This author has presented a model for Model Mappings based on formalisms. As a model is expressed in a given formalism, the transformation of a model into another model leads to

compare the formalisms they are based upon, i.e., to compare their primitives and their semantics. As a consequence, model mapping can be seen as a meta-modelling activity.

A widely used transformation language is the Extensible Stylesheet Language Transformation (XSLT) [23]. XSLT is the language used in XSL style sheets to transform XML documents into other XML documents. An XSL processor reads the XML document and follows the instructions in the XSL style sheet, and then it outputs a new XML document or XML-document fragment. Some authors, like Peltier [16], propose carrying out transformations using this language. He has defined a process called MTRANS to perform transformations. MTRANS uses XSLT to transform models, but in a more abstract way than XSLT. MTRANS is based on a meta-modelling approach, where a meta-model is used to define the semantics of each model. The MTRANS framework supplies a language and an environment to write models transformations. The language is composed by a set of instructions and a part depending on the particular used meta-model.

With regard to QVT (Query/View/Transformation) [18], as previously mentioned, it is a standard defined by the Object Management Group. Transformations in the context of QVT are classified into relations and mappings. On the one hand, relations allow verifying the consistence among related models. On the other hand, mappings implement transformations, in other words, they transform elements from one model to another. The ATLAS Transformation Language (ATL) [2] implements the QVT standard and it is composed of a mixture of imperative and declarative expressions. In the context of ATL, a source model is transformed into a target model by means of a set of transformations. Source and target models are specified by means of meta-models. ATL transformations are unidirectional, source model can only be read and target model can be modified.

Another proposal was developed by the Object Management Group: the Common Warehouse Metamodel Specification [5]. It specifies a model for describing transformations that introduces the concepts of black-box and white-box transformations. Both of these transformation styles only provide a relationship among model elements which are the sources and targets of a transformation, but do not express exactly what the resulting target will consist of. White-box transformations may have a procedure expression associated with the transformation, allowing for a program fragment in some implementation language to describe the implementation of the transformation.

Finally, there are authors who have defined a specific taxonomy, such as Mens [13]. By taxonomy, the author means “a system for naming and organizing things into groups which share qualities”. That taxonomy can help software developers in choosing a particular model transformation approach that is best suited for their needs; it can help tool builders to assess the strengths and weaknesses of their tools compared to other tools; and it can help scientists to identify limitations across tools or technology that need to be overcome by improving the underlying techniques and formalisms.

As conclusion, it is important to highlight the amount of proposals to express model-to-model transformations. Each proposal has advantages and disadvantages and there is not only one proposal more useful than others. The choice of one notation depends mainly on the analyst that will work with it. Comparing our proposal with existing ones, our work is not focused on a specific transformation language. This is the main contribution of our work with regard to existing proposals. This paper aims to propose a meta-model that is abstract enough to support the main concepts related to transformation rules in such a way that these concepts can be expressed independently of the particular syntax or notation of a specific transformation language. Once the transformation has been specified by means of the transformation model, the analyst can transform that model into a specific transformation language, according to analyst’s preferences.

USIXML AND ITS TRANSFORMATIONAL APPROACH

This section briefly presents UsiXML and its transformational approach, including a description of the current UsiXML transformation meta-model which is based on graph notation.

UsiXML [11] proposes a user interface development process which is compliant with MDE. The UsiXML approach proposes a user interface description language aimed at describing user interfaces with various levels of details and abstractions, depending on the context of use. UsiXML supports a family of user interfaces such as, but not limited to: device-independent, modality-independent, and context-independent.

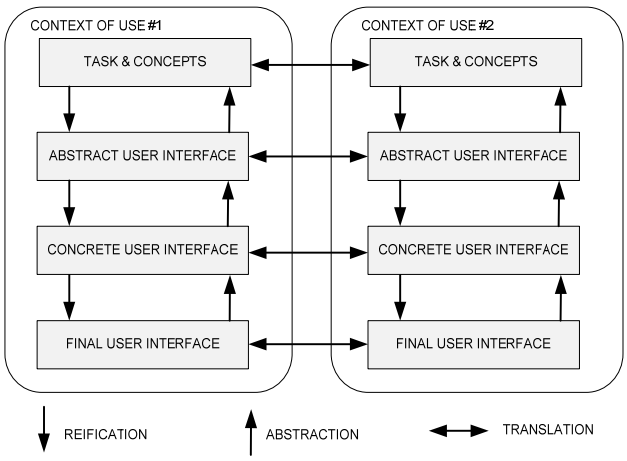

Figure 1. The four basic levels of abstraction of UsiXML and the three basic transformation types

UsiXML is structured according to four basic levels of abstraction defined by the Cameleon Reference Framework [3] (see Figure 1).

- **Task and Domain:** this level describes the various interactive tasks to be carried out by the end user and the domain objects that are manipulated by these tasks.
- **Abstract user interface (AUI):** this level provides a user interface definition that is independent of any modality of interaction (e.g. graphical interaction, vocal interaction, etc.).
- **Concrete user interface (CUI):** this level represents a user interface independently of any computing platform or programming toolkits peculiarities.
- **Final user interface (FUI):** in this level operational user interfaces are located, i.e., any user interface running on a particular computing platform either by interpretation or by execution.

Since the Task and Domain level is the more abstract level, and the FUI is the less abstract level, we can consider that these 4 abstraction levels are ordered and that there are 3 pairs of adjacent levels: 1) Task and Domain is adjacent to AUI; 2) AUI is adjacent to CUI; and 3) CUI is adjacent to FUI.

The Cameleon Reference Framework also exhibits three types of basic transformations (see Figure 1): (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 corresponding level of abstraction but aimed at a different context of use.

Previous works proposed a transformational approach to handle transformations between the different described abstraction levels and considering different interaction modalities and contexts of use [9, 10, 19, 20]. This transformational approach is based on graph transformation rules. Graph transformations are performed as follows. Let *G* be the initial UsiXML specification, when 1) a *Left Hand Side (LHS)* matches into *G* and 2) a *Negative Application Condition (NAC)* does not match into *G*, 3) the LHS is replaced by *Right Hand Side (RHS)*. *G* is consequently transformed into *G'* (the resultant UsiXML specification). All elements of *G* that are not covered by the match are left unchanged. This transformation approach is sustained by TransformiXML, a tool that allows the definition and application of transformation rules.

The current version of the meta-model that characterizes this transformational approach is defined in the UsiXML V1.8 Reference Manual [21]. Figure 2 illustrates this meta-model. As we can see in Figure 2, the current transformation meta-model incorporate concepts that are strongly related to graph transformation rules. Next, according to the definitions provided in [21], we briefly explain each class of this meta-model.

- **uiModel:** is the topmost superclass containing common features shared by all component models of a user interface.
- **transformationModel:** contains a set of rules enabling the transformation of one model, at a certain level of abstraction, into another, or to adapt a model for a new context of use.
- **developmentPath:** refers to the abstraction level where the transformation starts (Task and Domain, AUI, CUI or FUI), and the abstraction level where the transformation ends. The source and target levels can be non-adjacent.

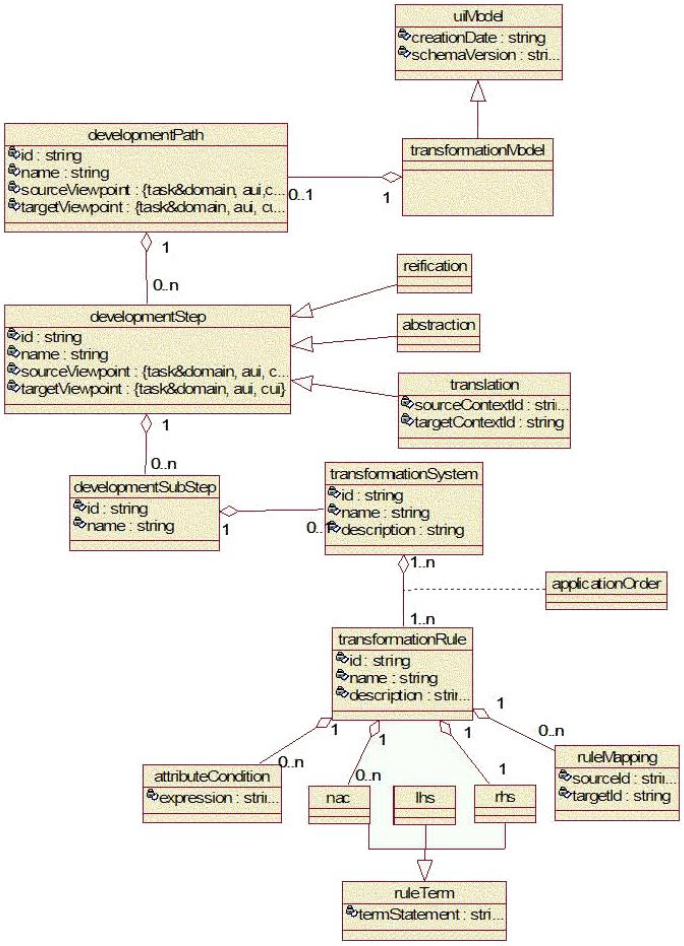

Figure 2. UsiXML transformation meta-model [21]

- **developmentStep:** is a transformation between adjacent abstraction levels, or a direct transformation between 2 contexts of use at the same abstraction level. There is a specialization of this class for each basic transformation type: reification, abstraction and translation.
- **developmentSubStep:** a development step can be composed of several sub-steps that need to be carried out

to accomplish a transformational goal. For instance, when transforming from Task and Domain to AUI (development step), the following sub-steps, or goals, were identified in [10]: 1) identification of AUI structure; 2) selection of abstract individual components; 3) spatio-temporal arrangement of abstract interaction objects; 4) definition of abstract dialog control; and 5) derivation of AUI to domain mappings.

- **transformationSystem:** is composed of a set of sequentially applied transformation rules that realize a basic development activity.
- **transformationRule:** performs a unitary transformation operation on a model. It is composed of a LHS, a RHS, and a NAC.
- **applicationOrder:** defines the order of execution of a transformation rule with respect to a particular transformation system.
- **attributeCondition:** textual expression indicating a condition scoping on element attributes of the LHS of a transformation rule.
- **ruleTerm:** is the content of a rule. It is composed of any fragment of uiModel.
- **nac:** NAC attached to a transformation rule.
- **lhs:** LHS of a transformation rule.
- **rhs:** RHS of a transformation rule.
- **ruleMapping:** defines the source and target models of a transformation rule.

This meta-model has been used as the basis for our proposal, as we explain in the next section.

A PROPOSAL FOR ENHANCING THE USIXML TRANSFORMATION META-MODEL

This section presents a transformation meta-model abstract enough to represent transformations independently of the underlying structure of a specific transformation language or technology. The proposed meta-model aims to enhance the current UsiXML transformation meta-model avoiding the use of explicit concepts related to graph-based transformation rules. Hence, our proposal has been based on the UsiXML transformation meta-model presented in [21]. On this basis, we have tried to eliminate explicit references to graphs concepts, and we have tried to add more expressiveness taking as inspirations features from ATL. We have also tried to provide clear explanations for all concepts that appear in our proposed meta-model.

Figure 3 presents the Transformation package of our proposed meta-model. The main concepts related to transformations are included in this package. Next, these concepts are explained.

Different *Models* can be created to specify an interactive system. For instance, we could use models of tasks, domain, AUI, CUI, context, etc., in order to specify an interactive system. All these models have some basic properties that were defined in [21].

A *Transformation Model* is a specialization of Model that has the aim to specify the rules that enable the transformation of a model at a certain level of abstraction, into another model at a different level. A Transformation Model also allows the adaptation of a model for a new context of use. When defining a Transformation Model, it is necessary to specify the initial abstraction level and the final abstraction level. These abstraction levels could be adjacent levels or not (see Figure 1). For instance, we could define a Transformation Model for transforming from the Task and Domain level to the adjacent AUI level, or for transforming from the AUI level to the non-adjacent FUI level. Furthermore, when defining a Transformation Model it is also necessary to specify the initial and final contexts of use. These contexts could be the same or different ones. For instance, we could define a Transformation Model for transforming from FUI models at context C1 to CUI models at context C1 or to CUI models at context C2. It is important to note that when considering different contexts of use, it is also meaningful to specify a Transformation Model with the same initial and final abstraction level, but for different contexts. For instance, we could define a Transformation Model for Transforming from CUI models at the C1 context to CUI models at the C2 context.

A Transformation Model aggregates an ordered set of *Transformation Steps*. A Transformation Step is a transformation between adjacent abstraction levels, or a direct transformation between two contexts of use at the same abstraction level. A Transformation Step is specified with a source abstraction level and a target abstraction level. Source and target must be adjacent abstraction levels, or the same level.

When the source abstraction level is higher than the target adjacent abstraction level, the Transformation Step is a *Reification*. For instance, we could define a Reification in which the source abstraction level is AUI and the target abstraction level is CUI. When the source abstraction level is lower than the target adjacent abstraction level, the Transformation Step is an *Abstraction*. For instance, we could define an Abstraction in which the source abstraction level is CUI and the target abstraction level is AUI. Another specialization of a Transformation Step is the *Translation*. A Translation allows the specification of a direct transformation, at a same abstraction level, from a source context of use to a target context of use. For instance, we could define a Translation in which the source abstraction level and the target abstraction level are CUI, the source context is C1, and the target context is C2. Thus, the ordered set of Transformation Steps that are chosen for a Transformation Model must be composed of Abstractions, Reifications and/or Translations that allow reaching the target abstraction level and the target context, from the source abstraction level and context.

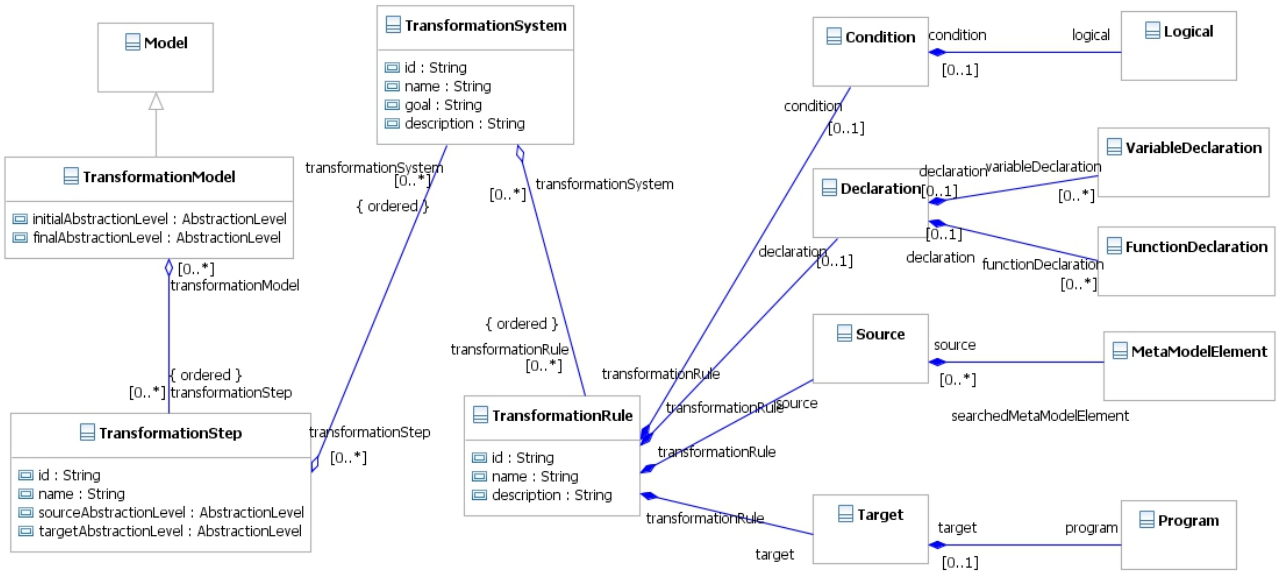

Figure 3. Transformation package of the new proposed transformation meta-model

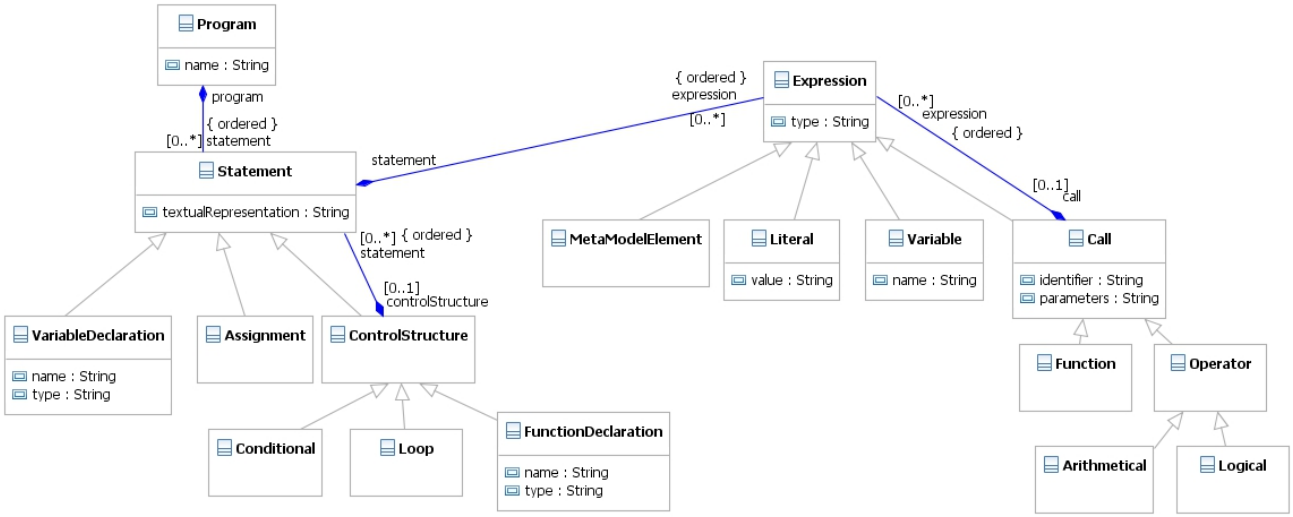

Figure 4. Program package of the new proposed transformation meta-model

Each Transformation Step is an ordered aggregation of *Transformation Systems*. A Transformation System groups a set of rules that accomplish a transformational goal. As an example, we have previously mentioned the goals that need to be reached when performing the transformation step from Task and Domain to AUI. Identification of the AUI structure and selection of abstract individual components are two of these goals. Each one of them corresponds to a transformation system.

Each Transformation System is an ordered aggregation of *Transformation Rules*. A Transformation Rule is a unitary transformation operation. For instance, the transformation system that corresponds to the goal of identifying the AUI structure is composed of several transformation rules, one of which generates an abstract container for each sub-task of the root task [19].

Regarding its structure, a transformation rule has an optional *Condition*. If the condition exists, it must be satisfied for the transformation rule to be applied. The condition is composed of a *Logical* expression that can be evaluated to true or false. A transformation rule can also have an optional *Declaration* section. In this section, auxiliary resources (e.g., variables, functions) can be specified in order to be used later in the specification of the rule. Hence, the declaration section can be composed of *Variable Declarations* and *Function Declarations*. Each transformation rule has a *Source* that represents a *Meta-Model Element* whose instances belong to the source model. Finally, each transformation rule has a *Target* that

represents what will be written in the target model. The target of a transformation rule is associated to a *Program*, which has enough expressivity to specify what needs to be written in the target model.

The previously mentioned concepts Logical, Variable Declaration, Function Declaration, and Program, are defined in the *Program package*, which is illustrated in Figure 4. This package represents a generic imperative programming language with which *Programs* can be written. A program is composed of an ordered set of *Statements*. *Variable Declaration* and *Assignment* are specializations of a statement which can not be composed of other statements. A variable declaration allows defining the name and type of a variable. An assignment allows specifying a value for a variable. *Control Structure* is another specialization of statement which can be composed of other ordered statements. Conditional, Loop, and Function Declaration are specializations of control structure. *Conditional* is a type of statement in which it is possible to specify a logical condition, a set of statements to trigger if the condition evaluates to true, and a set of statements to trigger if the condition evaluates to false. *Loop* is a type of statement in which it is possible to specify a logical condition and a set of statements that will be repeatedly executed while the condition evaluates to true. A *Function Declaration* allows specifying functions with a name and a return type. Furthermore, function declarations are composed of variable declarations that play the role of input parameters, variable declarations that play the role of local variables, and an ordered set of statements which compose the body of the function. The specific restrictions on the structure of a statement which is conditional, or loop, or function declaration, are specified by means of OCL constraints.

All type of statements can be composed of *Expressions* which return a value. Literal, Variable and Meta-Model Element are expressions which can not be composed of other expressions. A *Literal* expression returns its associated value. A *Variable* expression returns the value of a specific variable. A *Meta-Model Element* expression returns an object which is an instance of a meta-model element. *Call* is another specialization of Expression which can be composed of other ordered expressions. Function and Operator are specializations of Call. A *Function* call is an expression that returns the value which results from the execution of a specified function. In a similar way, an *Operator* call, (*Arithmetical* or *Logical*) is an expression that returns the value which results from the execution of a specified operator. Again in this case, the specific restrictions on the structure of the different types of expressions are specified by means of OCL constraints.

Even though there are relationships between elements of the Transformation package and elements of the Program package, the Program package is independent enough to be replaced, if necessary, by another Program package which represents other type of programs, for instance, declarative or functional programs.

Figure 5. MetaModel package of the new proposed transformation meta-model

Finally, the previously mentioned concept Meta-Model Element is defined in the MetaModel package, which is illustrated in Figure 5. In the context of UsiXML, a *Meta-Model Element* represents an element of any UsiXML meta-model published in [21]. The meta-model element is composed by a set of *Attributes*. Furthermore, *Relationships* can be established among meta-model elements.

AN ILLUSTRATIVE EXAMPLE

As an example, we are going to explain how the proposed meta-model can store transformation rules independently of the language used to carry out the transformations. Once the transformation is expressed according to the meta-model, all this information can be translated into a specific transformation language, such as the graph notation or ATL.

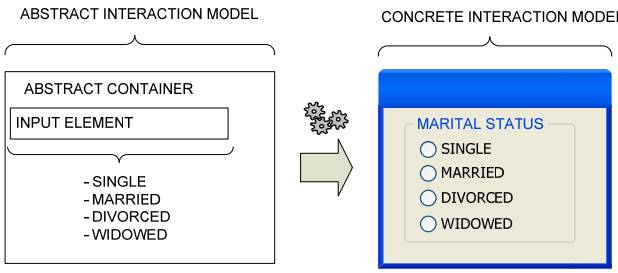

Figure 6. Example to instantiate the transformation meta-model

The goal of the example is to transform from the Task and Domain level to the CUI level. This aim is divided into two transformation steps: a transformation from the Task and Domain level to the AUI level, and a transformation from AUI level to the CUI level. For space reasons and to simplify the example, we focus on the last step. More specifically, we focus on a transformation rule to transform an input element with restricted values of an AUI model to a list of radio buttons in a CUI model, as Figure 6 shows. This transformation is useful, for instance, in a form where the user has to provide his/her marital status. Possible values are only: single, married, divorced and widowed.

Firstly, we are going to show how different instances of classes of our proposed meta-model store the required information to perform the transformation of the example.

- **TransformationModel**: it includes the set of transformation rules necessary to perform a transformation from the Task and Domain level to the CUI level. The attributes have the following values: *initialAbstractionLevel*: Task and Domain; *finalAbstractionLevel*: CUI.
- **ContextModel**: initial and final contexts are the same, there is no change of context.
- **TransformationStep**: in the example, this class has two instances of type *Reification.* One instance to represent the transformation from the Task and Domain models to the AUI model and another from the AUI model to the CUI model. With regard to the last instance, the attributes are: *id*: AUItoCUI; *name*: From AUI to CUI; *sourceAbstractionLevel*: AUI; *targetAbstractionLevel*: CUI.
- **TransformationSystem**: according to [10], the goals that need to be achieved when transforming from AUI to graphical CUIs are: 1) reification of abstract containers into concrete containers; 2) selection of concrete individual components; 3) arrangement of concrete individual components; 4) navigation definition; 5) concrete dialog control definition; and 6) derivation of CUI to domain mappings. Each one of these goals will have a corresponding transformation system instance. Our specific rule to transform an input element to a list of radio buttons is included in the second goal, selection of concrete individual components. The attributes of this transformation system instance have the following values: *id*: 4333e; *name*: Selection of concrete individual components; *goal*: Select the concrete individual components to be used; *description*: transforms abstract individual components to concrete individual components.
- **TransformationRule**: this instance stores the rule to transform an input element to a list of radio buttons. The values of its attributes are: *id*: Rdbtt1; *name*: Input element to radio button; *description*: transforms an input element with a limited number of possible values into a list of radio buttons.
- **Condition**: only input elements with a set of limited possible values can be transformed into radio buttons. In the example, this instance is related to a logical expression that verifies if the input element has a set of limited possible values.
- **Source**: this instance is used to specify which element of the AUI model is going to be transformed. In this example, it is an input element from an abstract container.
- **Target**: this instance stores a program that specifies how to build the target element of the transformation. In this example, it is a program that builds a list of radio buttons of the CUI model.
- **MetaModelElement**: instances of this class represent all the classes of the AUI meta-model and the CUI meta-model involved in the transformation. Classes affected of the AUI meta-model are: *Abstract Container; Abstract Individual Component; Facet; Input; Selection Value.* With regard to the CUI meta-model, affected classes are: *Window; Tabbed Dialog Box; Radio Button.*
- **Attribute**: each instance of this class represents an attribute of the classes belonging to the AUI meta-model or to the CUI meta-model included in the transformation. For example, the class *Selection Value* of the AUI meta-model has an attribute called *name*.
- **Relationship**: instances of this class store the relationships among classes of the AUI meta-model or the CUI meta-model included in the transformation. For example, the class *Input* is related to the class *Selection Value* in the AUI meta-model.
- **Program**: this instance represents the set of statements of the transformation process that are used to reach the target.
- **Statement**: this class is specialized in other ones depending on the type of statement. In our example, there are assignments to perform the transformation from an input element to a RadioButton.
- **Expression**: this class is specialized in other classes to represent sentences that return a value. For example, there is an instance of the class *MetaModelElement* for each element of the CUI model created in this transformation.

All the information stored in a transformation model compliant with our proposed transformation meta-model can be translated into an existing transformation language to perform the transformation, such as, the graph notation or ATL.

Regarding the graph notation, from the information available in the previously explained transformation model example, we could automatically generate the corresponding graph-based transformation rules defined in [19]. Figure 7 illustrates the NAC of the transformation rule. Figure 8 illustrates the LHS, and Figure 9 illustrates the RHS. (In Figure 7, Figure 8, and Figure 9, the red part corresponds to the transformation to a graphical CUI model).

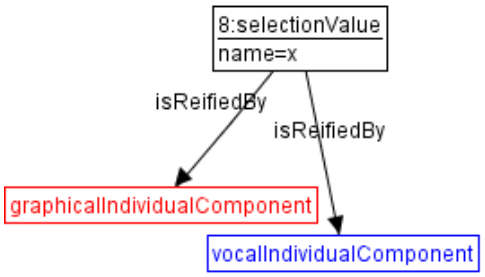

Figure 7. NAC statement to transform an input element to a radio button with possible options [19]

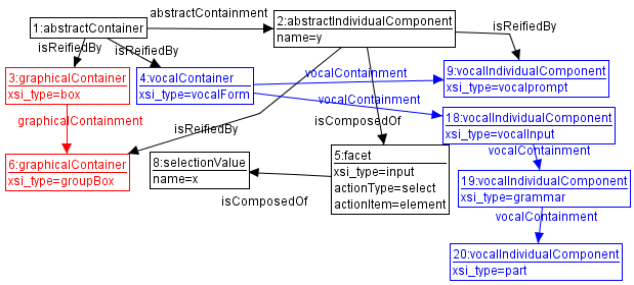

Figure 8. LHS statement to transform an input element to a radio button with possible options [19]

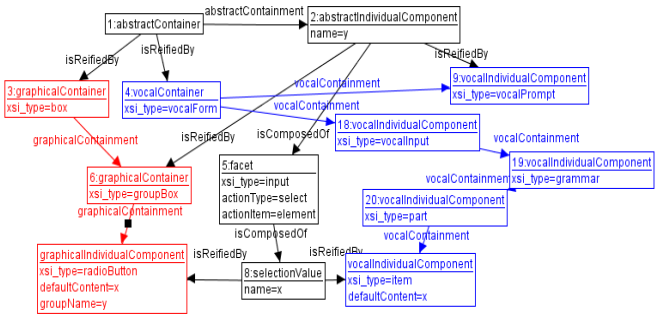

Figure 9. RHS statement to transform an input element to a radio button with possible options [19]

From the information available in the previously explained transformation model example, we could also automatically generate the following corresponding ATL code:

```
module TransformationRadioButton;
create OUT: CRadioButton from IN: AInoutElement;
uses strings;
rule container2window{
        from
        c:AInoutElement!abstractContainer
        to
        out:CRadioButton!window (
        title <- c.title)
}
rule input2TabbedDialogBoxes{
        from
        ie:AInoutElement!Input(ie.type.oclIsKindOf(AInoutElement!SelectionValue))
        to
        out: CRadioButton!TabbedDialogBoxes(
        name <- ie.name)
}
rule selectionValue2RadioButton{
        from
        sv : AInoutElement!SelectionValue (
        sv.type.oclIsKindOf(AInoutElement!Input))
        to
        out: CRadioButton!RadioButton(
        value <- sv.name)
        }
```

This example suggests that our proposed meta-model is abstract enough to represent concepts of transformation languages as dissimilar as ATL and graphs. Analysts can put all their efforts on specifying the transformation rules independently of transformation language details, and in a next step, these rules can be translated into a specific transformation language that performs the transformation automatically.

CONCLUSION

This paper proposes a new transformation meta-model independent of graph transformations for a user interface description language called UsiXML. The aim is to provide a transformational approach independent of the specific syntax and notations of transformation languages or technologies. This paper is a first step to reach the aim. Once the transformation meta-model has been defined, a precise syntax must be specified to write transformation rules. The transformation meta-model expresses only how to store transformation rules, but the analyst must have an unambiguous language to represent those rules. This syntax is going to be defined as a future work.

Furthermore, once the transformation meta-model supports all type of transformations and a precise syntax has been defined to specify them, the last step will be to specify different translators. These translators must translate all the information stored in the transformation model into a transformation language selected by the analyst. Each transformation technology will have a specific translator that performs the translation automatically taking as input a transformation model.

The paper has exemplified how a transformation model stores the required information to translate transformation rules into a graph-based notation or ATL. As future work, we are going to test with other transformation technologies such as XSLT or CWM. Moreover, we have not tested the transformation meta-model to assess whether or not it supports model-to-code transformations. This kind of transformations is very common in UsiXML because final

user interfaces are automatically generated from CUI models. Hence, it would be very interesting that analysts could also specify this type of transformations by means of the transformation model.

ACKNOWLEDGMENTS

This work has been developed with the support of the ITEA2 Call3 UsiXML project under reference 2008026. It also has the support of MICINN under the project SESAMO (TIN2007-62894) and has been co-financed by ERDF.

REFERENCES

1. Assmann, U.: How to uniformly specify program analysis and transformation with graph rewrite systems. 6th International Conference on Compiler Construction, Vol. LNCS 1060. Springer Berlin / Heidelberg (1996).
2. ATL: http://www.eclipse.org/m2m/atl/. Last visit: March 2010
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):289–308, 2003.
4. Caplat, G., Sourrouille, J.L.: Model Mapping in MDA. Workshop in Software Model Eng. (WISME) (2002).
5. CWM Partners. Common Warehouse Metamodel (CWM) Specification. OMG Documents: ad/01-02-{01,02,03}, Feb. 2001.
6. Gogolla, M.: Graph Transformations on the UML Metamodel. ICALP Workshop Graph Transformations and Visual Modelling Techniques, Waterloo, Canada (2000).
7. Ho, W.M., Jézéquel, J.-M., Guennec, A.L., Pennaneach, F.: UMLAUT: an Extendible UML Transformation Framework. In: Tyugu, R.J.H.a.E. (ed.): 14th IEEE International Conference on Automated Software Engineering, ASE'99. IEEE (1999).
8. Karsai, G., Agrawal, A., Shi, F., Sprinkle, J.: On the use of graph transformation in the formal specification of model interpreters. Journal of Universal Computer Science, Vol. 9 (2003) 1296–1321.
9. Limbourg, Q. and Vanderdonckt, J. Transformational development of user interfaces with graph transformations. In R. J. K. Jacob, Q. Limbourg, and J. Vanderdonckt, editors, *CADUI*, pages 105–118. Kluwer, 2004.
10. Limbourg, Q. *Multi-Path Development of User Interfaces*. PhD thesis, Universithé catholique de Louvain, November 2004.
11. Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., and López-Jaquero, V. USIXML: A Language Supporting Multi-path Development of User Interfaces. In R. Bastide, P. A. Palanque, and J. Roth, editors, *EHCI/DS-VIS*, volume 3425 of *Lecture Notes in Computer Science*, pages 200–220. Springer, 2004.
12. Mellor, S.J., Clark, A.N., Futagami, T.: Guest Editors' Introduction: Model-Driven Development. IEEE Software, Vol. 20 (2003) 14-18.
13. Mens, T., Gorp, P.V.: A Taxonomy of Model Transformations. Language Engineering for Model-Driven Software Development - Dagstuhl Seminar Proceedings, Dagstuhl, Germany (2005).
14. MOF: http://www.omg.org/spec/MOF/2.0/. Last visit: March 2010.
15. Pastor, O., Molina, J.: Model-Driven Architecture in Practice. Springer, Valencia (2007)
16. Peltier, M., Bézivin, J., Guillaume, G.: MTRANS: A general framework based on XSLT for model transformations. Workshop on Transformations in UML, WTUML'01, Genova, Italy (2001).
17. Pérez-Medina, J. L., Dupuy-Chessa, S., and Front, A. A Survey of Model Driven Engineering Tools for User Interface Design. In M. Winckler, H. Johnson, and P. A. Palanque, editors, *TAMODIA*, volume 4849 of *Lecture Notes in Computer Science*, pages 84–97. Springer, 2007.
18. QVT: http://www.omg.org/spec/QVT/1.0/. Last visit: March 2010.
19. Stanciulescu, A. *A Methodology for Developing Multimodal User Interfaces of Information Systems*. PhD thesis, Université catholique de Louvain, June 2008.
20. Stanciulescu, A., Limbourg, Q., Vanderdonckt, J., Michotte, B., Montero, F.: A Transformational Approach for Multimodal Web User Interfaces based on USIXML.: ICMI. ACM Press (2005) 259-266.
21. Université catholique de Louvain. UsiXML V1.8 Reference Manual. 2008. Available at: http://www.usixml.org/index.php?mod=pages&id=5. Last visit: March 2010.
22. Varró, G., Schürr, A., Varró, D.: Benchmarking for graph transformation. IEEE Symposium on Visual Languages (VL/HCC 2005), University of Texas at Dallas (2005).
23. XSLT Specification: http://www.w3.org/TR/xslt. Last visit: March 2010.