

UNIVERSITE CATHOLIQUE DE LOUVAIN

FACULTE DES SCIENCES APPLIQUEES



A Comparative Analysis of Transformation Engines for User Interface Development

A thesis submitted in fulfillment of the requirement
for the degree of "Licence en Informatique" of the
Université catholique de Louvain

By Jean-Pierre DELACRE

Committee in charge:

Prof. Jean Vanderdonckt, Supervisor

Adrian Stanculescu, Reader

Prof. Tom Mens, Reader

Academic year 2006-2007

A Comparative Analysis of Transformation Engines for User Interface Development.....	0
Chapter One. Introduction.....	6
1.1 Context.....	6
1.2 Hypotheses.....	7
1.3 Primary goal	7
1.4 Reading Map	8
Chapter Two. Model-Driven Engineering of User Interfaces: state of the art.....	10
2.1 Definitions.....	10
2.2 Transformation Engines	14
2.3 Selected transformation Engines and meta-model	15
2.3.1 Meta-model	15
2.3.2 Transformation Engines	16
2.3.1.1 ATOM ³	16
2.3.1.2 Custom java application.....	17
2.3.1.3 ATL.....	17
Chapter Three. ATOM ³	18
3.1 Implementation	19
3.2 Meta-model	20
3.3 Transformations.....	22
3.3.1 The rules sets	23
3.3.2 The rules.....	23
3.3.3 Properties of the objects.....	27
Chapter Four. Custom transformation engine in java	32
4.1 Implementation	32
4.2 Details of the implementation.....	34
4.2.1 The graphical user interface (UsiXMLTransformationGui)	35
4.2.2 The main class.....	37

4.2.3 The Rules class	39
Chapter Five: ATL.....	47
Chapter Six. Case studies	49
6.1 Implemented rules.....	49
6.2 Use case: Currency convertor	50
Step 2: From Task a to AUI Model	51
Sub-step 2.1: Rules for the identification of AUI structure.....	51
Sub-step 2.2: Rules for the selection of AICs	52
Sub-step 2.3: Rules for spatio-temporal arrangement of AIOs	52
Sub-step 2.4: Rules for the definition of abstract dialog control.....	52
Sub-step 2.5: Rules for the derivation of the AUI to domain mappings.....	52
Step 3: From AUI Model to CUI Model	52
Sub-step 3.1: Reification of AC into CC.....	52
Sub-step 3.2: Selection of CICs.....	52
Sub-step 3.3: Arrangement of CICs.....	53
Sub-step 3.4: Navigation definition	53
Sub-step 3.5: Concrete Dialog Control Definition.....	53
Sub-step 3.6: Derivation of CUI to Domain Relationship.....	53
6.2.1 Currency convertor in AToM ³	54
6.2.2 Custom java transformation engine	60
6.3 Use case: Polling System	64
Step 1: The Task and Domain Models.....	64
Step 2: From Task and Domain Models to AUI Model.....	68
Sub-step 2.1: Rules for the identification of AUI structure.....	68
Sub-step 2.2: Rules for the selection of AICs	68
Sub-step 2.3: Rules for spatio-temporal arrangement of AIOs	69
Sub-step 2.4: Rules for the definition of abstract dialog control.....	69
Sub-step 2.5: Rules for the derivation of the AUI to domain mappings.....	69
Step 3: From AUI Model to CUI Model	69
Sub-step 3.1: Reification of AC into CC.....	69
Sub-step 3.2: Selection of CICs.....	69

Sub-step 3.3: Arrangement of CICs	70
Sub-step 3.4: Navigation definition	70
Sub-step 3.5: Concrete Dialog Control Definition.....	70
Sub-step 3.6: Derivation of CUI to Domain Relationship.....	70
6.3.1 AToM ³	71
6.3.2 Custom java transformation engine	79
Chapter 7: Comparison of the four techniques	87
7.1 Summaries	87
7.1.1 AToM ³	87
7.1.2 ATL.....	88
7.1.3 Custom transformation engine in java	88
7.2 Comparison	89
7.2.1 Tools-specific issues	89
7.2.1.1 AToM ³	89
7.2.1.1.1 Meta-model and models creation	89
7.2.1.1.1 Rules creation	89
7.2.1.2 Custom java transformation engine	90
7.2.1.2.1 Meta-model creation	90
7.2.1.2.2 Model creation.....	90
7.2.1.2.3 Rules creation	91
7.2.2 Tools-specific advantages	91
7.2.2.1 AToM ³	91
7.2.2.2 Custom java transformation engine	92
7.2.3 Global comparison	92
7.2.3.1 Implementation paradigm and required programming skills.....	93
7.2.3.2 The model-to-model approach	93
7.2.3.3 Code generation.....	94
7.2.3.4 Pattern matching	94
7.2.3.5 Rules scheduling and organization, inheritance	94

7.2.3.6 Bi-directionality.....	95
7.2.3.7 Performance	95
7.2.3.8 Flexibility and maintainability	95
7.2.3.9 Completeness.....	96
7.3.3 Conclusion.....	98
Bibliography	100
Appendix A.....	103
2.3.1.1 Task Model.....	103
2.3.1.2 Domain Model	104
2.3.1.3 Abstract User Interface Model.....	105
2.3.1.4 Concrete User Interface Model.....	109
2.3.1.5 Final User Interface	110
2.3.1.6 Context Model	110
2.3.1.7 Mapping Model.....	111
2.3.1.8 Transformation Model	112
Appendix B	114

Chapter One. Introduction

1.1 Context

Cobol, Visual Basic, C/C++, Java, XML, .net, and many other languages, spread over many OS like Windows, Unix, MacOS, etc: today's opulence in computer science leads to an all-time growing complexity in software engineering. With so many technologies and platforms, a lot of issues are raised, among those: reusability, interoperability, portability, adaptability.

Reducing the number of those technologies and platforms could be a solution. But the chance for it to happen is negligible, and there is no technology that prevails on the others.

For that reason, another solution is needed. The object-oriented approach was a first attempt to decrease the complexity of software engineering, but it proved insufficient.

In this context came the Model Driven Engineering, with an ambitious objective: moving the development effort from implementation to solution modeling. The MDE refers to the systematic use of models as first and primary development step. We create a model for our application that is totally independent of any technology, in a more natural language than programming language. Then, transformation rules are applied on this model to give the wanted implementation.

The advantages are obvious:

- Reusability and portability are greatly improved: the model stays valid for each platform or technology.
- Complexity is decreased: the model can be expressed in a more natural language.
- As a result of reduced complexity, the quality of the applications can be improved, and the number of error reduced.

The OMG (Object Management Group), because of its strength and influence, was in a good position to create a standard for MDE. As we can see on their site [OMG], "OMG Task Forces develop enterprise integration standards for a wide range of technologies, and an even wider range of industries. OMG's modeling standards enable powerful visual design, execution and maintenance of software and other processes". The OMG tries to cope with the problem of the lack of systematization and reusability by offering a way of creating programs independently of the context of use.

Starting from the MDE, the OMG created, in 2000, the MDA [MDA, 6] (Model Driver Architecture). The MDA provides a systematic framework to develop software using engineering methods and tools. Its core infrastructure is composed of several OMG standards, including UML (Unified Modeling Language), XMI (XML Metadata Interchange) and the MOF (Meta-Object Facility). The MDA offers standards to design models, independently of the

platform or technology, which are expressed in a more human-friendly language (like UML or another MOF-compliant meta-model).

MDA's principle is to create Platform Independent Models (PIM) that will then be transformed into Platform Specific Models (PSM), which will be used to generate the application code. PIM are thus reusable for every platform; this represents a great progress in terms of interoperability and productivity and, because the language used to create models is less complex, the risks of errors from the user interface designer is decreased and the maintenance becomes easier.

The interoperability problem obviously also affects the engineering of Graphical User Interfaces (GUI's). The problem not only comes from the number of different contexts of use, but also from the way graphical interfaces are created. Up to recently, there were two ways of creating an interface; the first, old-fashioned method, is the programmatic approach: very slow, and the result is of variable quality due to a high complexity. The second method is using visual generators or descriptive language editors, which means a highly simplified and fastened creation process compared to the programmatic approach.

But even the second approach does not resolve the problem of reusability: there is still the need of "manually" modifying the interface of each context of use; here too, the MDA allows great benefits in terms of productivity and reusability.

1.2 Hypotheses

Because they represent the biggest part of the interactive software today, we will restrict ourselves to the information systems.

Here we talk only about HCI (Human-Computer Interfaces), and only the graphical part of it (we won't take the vocal, 3-D or multimodal interfaces into consideration) because, once again, it represents most of the HCI.

So, we will study some transformation engines that focus on the conception of Graphical User Interfaces (GUI) on information systems. And to do this, we will use several examples, mainly an example of simple web interface, for which we will try to implement the rules in some transformation engines.

There exist several user interfaces description languages, so we had to choose one. We will see in the chapter 2 which language we chose as meta-model, and why.

1.3 Primary goal

The Model Driven Architecture (and model driven engineering in general) is much based on model-to-model transformations, and there exist a lot of tools to create these. The graphical user interface designer has to choose one, but these tools are not all equal, or even decide to

create a custom one that could better suit his needs. Existing tools can be very different with respect to their transformation approach (graph on programmatic syntax), implementation paradigm (declarative, imperative, or both), and many other points. Some are easier to use, while others allow the creation of more complex rules. Moreover, most of them have their own programming language.

So, following the needs of the interface designer, his skills in programming and others criteria, the choice of creation tool will change, and therefore, it is very important to know the differences between the known techniques, and to see which one is more adapted for each case.

What we will do is, focusing on graphical interfaces development, review some existing transformation engines, and create ourself a custom one, implemented in java. And we will make a comparative analysis of these transformation engines.

1.4 Reading Map

In chapter 2, we will describe the state of the art in transformation engines, as well as our choices and why we made them.

- In chapter 2.1 we define all the terms relative to the transformation process.
- In chapter 2.2 we give a short list of existing transformation engines and describe briefly which ones we chose.
- In chapter 2.3 we explain the meta-model which meta-model we chose, and why. We also describe the transformation engines we selected as well as the reasons why we selected those.

Then, chapters 3, 4 and 5 are devoted to the description of the transformation engines we selected. Chapter 5 is shorter than the two others, because the transformation engine we describe in it hasn't been used by us, but by another student, that has given us a detailed description of it, for the sake of the comparison.

Chapter 6 is a detail description of an example of interface created in the transformation engines we selected.

Chapter 7 is the comparison between the transformation engines.

- In chapter 7.1 we give a short summary of the three transformation engines we selected.
- In chapter 7.2 we give the tools-specific issues we encountered.
- In chapter 7.3 we enumerate the tools-specific advantages we notes during the utilization of them.

- In chapter 7.4 we make a global comparison of the tools we used.
- Finally, in chapter 7.5 we give our conclusion and discuss the result.

Chapter Two. Model-Driven Engineering of User Interfaces: state of the art

2.1 Definitions

MDE, *Model Driven Engineering*: a problem and its solution are modeled through different levels of abstraction that allows hiding the complexities of a platform. Instead of coping with all the subtleties of a platform, the software engineer handles simpler models. The models are conform to a single *meta-model*, to allow tools automatically dealing with the models, for example transforming a model into another (*model transformation*).

OMG, *Object Management Group*: Consortium originally created for the setting of standard in object-oriented systems. It is now focused on modeling. OMG is the initiator of the MDA.

MDA, *Model driven Architecture*:

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.

So, MDA is an application of the MDE. The goal of the MDA is to create Platform Independent Models (PIM) that can be later refined into Platform Specific Models (PSM) through transformations rules. Finally, the final code will be generated from the PSM.

A model can represent many things; it is an abstract representation of real world concepts. A model is a formal representation of data, functions or application behavior. The UML representation of an application is a model, as well as the

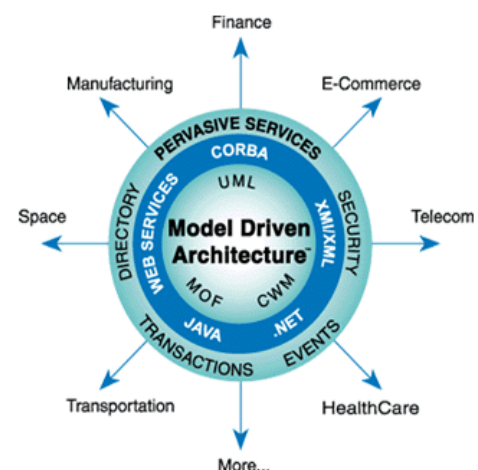


Figure 2-1 The MDA architecture

code of the application. A model is simpler than the real world concept it represents; it is designed for a specific purpose, and allows us to use a concept without having to cope with its real world complexity.

The MDA is a standard approach of modeling and automated mapping between the models. The “kernel” of the MDA is a stack of four modeling levels described in Figure 2-2 [Nauwenko]

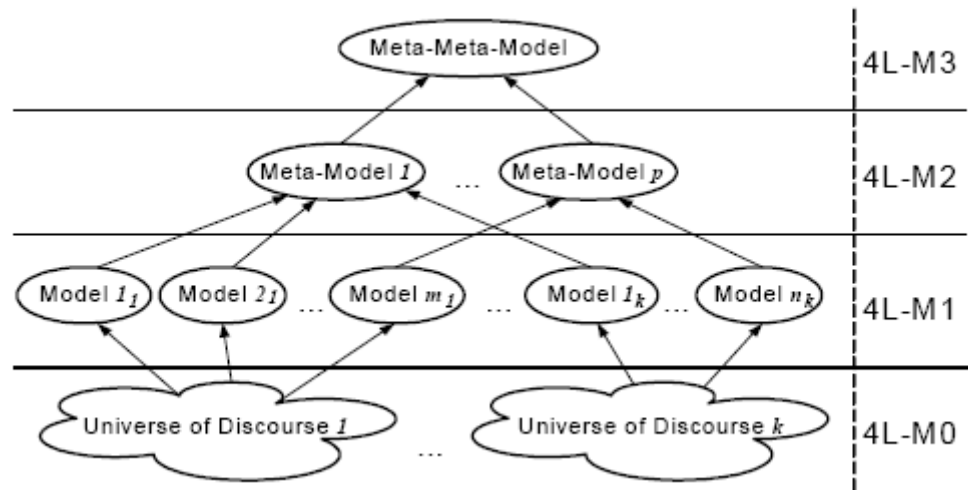


Figure 2-2 The four levels of MDA

- The level M4 is the meta-meta-model, that is, the model to which all other models have to conform.
- The level M3 represents the meta-models created for a specific business.
- The level M2 corresponds to the platform-specific implementation. So it can be, for example, a java program.
- Finally, the level M1 is some kind of “user-specific” data, like an execution of a platform-specific model of level M2.
- Of course, other levels can be added between the existent ones, to refine them.

To allow systematization, tools must be able to manipulate the models. The meta-meta-model is some sort of “legend” for all the models. A map uses only what is defined in its legend and so does a model with respects to its meta-model. So, with a single meta-meta-model, standardization is possible and we can create tools to manipulate the models.

UML is a possible meta-meta-model, but it isn’t the only one. So, the OMG created the MOF (Meta-Object Facility) [MOF]. In figure 2-2, the MOF is the level M4. It is a standard addressing meta-models and their manipulation. So, every model used as a meta-model should conform to the MOF.

And the OMG also provided a way of creating MOF-compatible models: the XMI recommendation [XMI] (see XMI definition further). Basing on the W3C XML standard, it offers a widely available tool on which the designers can map their meta-models.

By creating a meta-model that is independent of the used platform or technology, we make it possible to create model that are also independent of the platform or technology. Then we have to create a meta-model that is specific to the platform/technology. And because all the models conform to the MOF, we can create tools that transform the Platform Independent Model (PIM) into Platform Specific Models (PSM). These tools are based on Model Transformations (model transformations are explained further).

The OMG also defined a standard for the model-to-model transformations: the Query/View/Transformations (QVT) [QVT], which is in fact a meta-model of the transformations.

XMI, XML Metadata Interchange [9]: XMI is an OMG standard that maps the MOF to the W3C's eXtensible Markup Language (XML) [XMI]. XMI defines how XML tags are used to represent serialized MOF-compliant models in XML. MOF-based metamodels are translated to XML Document Type Definitions (DTDs) and models are translated into XML Documents that are consistent with their corresponding DTDs.

XMI solves many of the difficult problems encountered when trying to use a tag-based language to represent objects and their associations. Furthermore, the fact that XMI is based on XML means that both metadata (tags) and the instances they describe (element content) can be packaged together in the same document, enabling applications to readily understand instances via their metadata. Communication of content is both self describing and inherently asynchronous. This is why XMI-based interchange is so important in distributed, heterogeneous environments.

PIM, *Platform Independent Model*: model at a high level of abstraction that is independent of the any platform. It allows creating software more easily, and the result is reusable for each different platform.

PSM, *Platform Specific Model*: model at a low level of abstraction the is specific to a given platform.

Transformation Rule: given a model, a transformation rule gives another model. Transformations rules can be created by two means:

- Graphically: it is then composed of at least a Left Hand Side (LHS), a Right Hand Side (RHS) and a precondition (the opposite of the precondition is sometimes called NAC). The rule searches the LHS in the graph (by pattern matching) and replaces it by the RHS, is the precondition is verified.
- Programmatically: an algorithm browses the model and modifies it. This is a much harder method to create a transformation rule, but there are nearly no limitations, compared to the graph transformation.

MOF, Meta-Object Facility: OMG's standard describing meta-models and their manipulation.

This thesis will focus on automatic transformation of models, using transformation engines. We will see how we can create transformation rules and execute them either in an existing transformation engine, or by creating one. There exist a lot of transformation engines today, so we had to make a choice. This choice and the reasons why we made it will be explained further.

Finally, once the meta-model and transformation engines are chosen, and the transformation rules implemented, we will show how they apply on example use cases.

2.2 Transformation Engines

There exist today a lot of model-to-model transformation engines, which can be classified following six categories [Czarnecki]:

- Direct-manipulation approach
- Relational approach
- Graph-transformation-based approach
- Structure-driven approach
- Hybrid approach
- Other approaches

And the tools of each category can further be classified using other criteria, such as rules application scoping, rules scheduling and organizing, bi-directionality of the rules and many others. This really means a great number of different tools.

VIATRA [VIATRA], ATOM³ [ATOM], GreAT [GReAT], UMLX [UMLX], and BOTL [BOTL] are example of transformation engines; but there are many others.

Obviously, we couldn't use all of these tools, and our objective was to make a comparative analysis of transformations engines based on the implementation of rules into them.

The next chapter first explain which meta-model we chose, and why. Then, it explains which transformation engines we chose, and the reasons for it.

2.3 Selected transformation Engines and meta-model

2.3.1 Meta-model

To implement rules in transformation engine, we have to choose a meta-model (the specification language) to use for the sake of the comparison.

There exist several languages designed for the creation of user interfaces, such as XUL (www.mozilla.org/projects/xul/), UIML (www.uiml.org), XAML (www.xmls.net), XIML (www.ximl.org) and UsiXML (www.usixml.org).

All the transformation rules created for this thesis are based on the UsiXML (User Interface eXtensible Markup Language) programming language.

UsiXML is a User Interface Description Language (UIDL) that allows the specification of various types of UIs, based on the Cameleon reference framework for multi-target UIs.

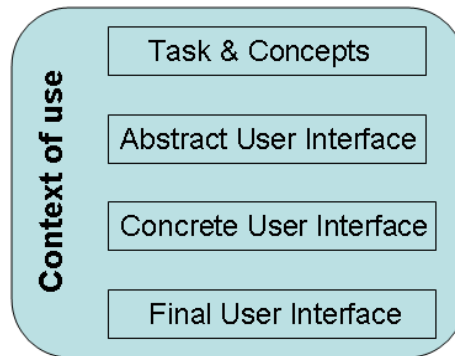


Figure 2-3 The Cameleon reference framework for multi-target UIs

This framework is itself based on four abstraction levels: the Task & concepts level, the abstract user interface level, the concrete user interface level and the final user interface level (cf. figure 2-3).

UsiXML is thus based on a transformational approach: the goal of this language is to allow creating an interface by only giving the tasks of the interface, and the objects it manipulates, independently of the targeted platform. Then, by means of model-to-model transformations rules, an abstract interface is created and, with other rules, transformed in a concrete interface that will serve to generate a final user interface.

Moreover, the underlying formalism of UsiXML is represented under the form of a graph-based syntax, thus well suited for the use with a tool like AToM³, using graphical representation of the meta-model and transformation rules.

We chose UsiXML as a meta-model for several reasons:

- UsiXML has been proven to be MDA-compliant [7]. It is structured according to the four basic levels of abstraction defined by the Chameleon reference framework, and based on a transformational approach (under the form of a graph-based graphical syntax), which is the main subject of this thesis.
- The underlying unique structure, based on a graph syntax, which is very easy to represent in graphical tools like AToM³ and ATL (in ATL, meta-model can be constructed graphically).
- Transformation rules are expressed conforming to the same meta-model as the transformed models.

UsiXML is composed of eight levels: Task Model, Domain Model, Abstract User Interface Model, Concrete User Interface Model, Final User Interface, Context Model, Mapping Model and Transformation Model. For a complete description of the UsiXML user interface description language, see the Appendix B.

2.3.2 Transformation Engines

2.3.1.1 AToM³

First, we will use a program specifically designed to model transformation rules, named AToM³. AToM³ belongs to the “graph-transformation-based approach” category; we chose it because graph transformations are easy to implement, and the subject of this thesis is to compare transformation engines and see if the increased flexibility of solutions like AToM³ (that uses the declarative paradigm for the implementation of the transformation rules) involves a decreased powerfulness and completeness compared to harder-to-use, imperative solutions.

In AToM³, we create, graphically, one or several meta-model(s) and model-to-model transformation rules that apply on this (these) meta-model(s).

Transformations are coded using graph transformation rules. That is, a transformation is composed of two graphs (LHS, RHS) and optional condition and action:

- The LHS (Left Hand Side) is a sub-graph that will be found in a graph using the *pattern matching* method.
- The RHS (Right Hand Side) is the sub-graph that will replace the LHS in the graph.
- The condition (name of the pre-condition in AToM³) is coded in python (<http://www.python.org/>), and must return true for the graph transformation rule to be executed.
- The action, also coded in Python, is executed after the graph transformation rule, if this one has been properly executed.

...

2.3.1.2 Custom java application

Then, we will try something else by creating our own program and interface, using the java programming language. It is thus quite the opposite of AToM³. Instead of creating graphs, that are quite intuitive, here all the transformation rules are coded in java. This means a longer and more fastidious work (we will see further that it really is hard to implement, at least the way we made it) as well as the need of good programming skills. The comparison between the two should allow us to tell in this increased complexity is really rewarding in terms of performance and completeness of the transformation engine.

The program first reads a UsiXML file and translates it into java objects. Then the transformation rules apply on these objects and, finally, translate the objects back into UsiXML.

2.3.1.3 ATL

The third techniques added for the comparison, is using the ATL plug-in for eclipse to create transformation rules. ATL can be considered as somewhere between AToM³ and java, as transformation rules are coded (in the ATL specific language), but the meta-model can be graphically designed.

ATL is presented in this thesis because it belongs to the “Hybrid approach” category. ATL is using declarative AND imperative constructions. It is obviously more flexible than java and the models and rules created in it are far more easily implemented and maintained than in java. But, unlike AToM³, it is not fully declarative, and allows more complex constructions thanks to the imperative possibilities. ATL is “in-between” AToM³ and java.

The choices we made allow us to have three very different kinds of transformation engines, well representative of the three categories they belong to. We will describe the three techniques, and then, based on two examples, we will try to do a comparison and explain the pros and cons of each technique.

In the next three chapters, we give the detailed description of each of the transformation engines we used.

Chapter Three. ATOM³

AToM3 is a tool for multi-paradigm modeling that is used for meta-modeling and model-transforming. As we can read on AToM³'s site¹ : “Meta-modeling refers to the description, or modeling of different kinds of formalisms used to model systems” and “Model-transforming refers to the (automatic) process of converting, translating or modifying a model in a given formalism, into another model that might or might not be in the same formalism”.

AToM3 is a graph transformation tool. This means that formalisms and models are described as graphs, and the transformations themselves are declaratively expressed as graph-grammar models.

First, we modeled UsiXML in AToM³ as a meta-model. As we could not represent completely UsiXML, we chose the classes concerned by the examples explained further². Instead of creating multiples meta-model (i.e. for tasks&concepts model, abstract user interface model, etc.), we created only one to represent everything.

AToM³ works as follows: a source model (graph) is submitted to transformations rules that modify it to build the target model.

The source and target models each conform to a meta-model (the source and target meta-model can be distinct or grouped in a single meta-model), and the transformation rules use pattern conforming to source and target meta-models.

In AToM³, no distinction is made between source and target model. The source model itself is transformed by the transformation rules.

¹ <http://atom3.cs.mcgill.ca/>

² « Polling System », p.8 and « Conversion system », p.8

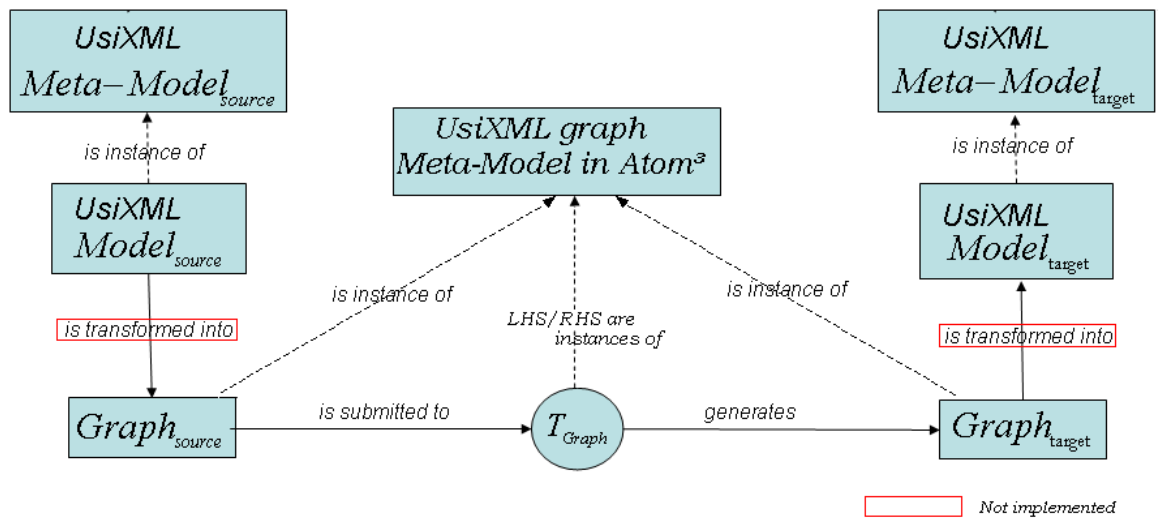


Figure 3-1 ATOM³ execution engine

3.1 Implementation

All the transformation rules we implemented in ATOM³ are described in our rules catalog (see Appendix B). The complete documentation of the ATOM³ rules can be found in the file Rules_ATOM³.pdf on the CD joint to this thesis.

The transformations are expressed as graph-grammar models (a graph grammar is the combination of a set of graph rewriting rules and the graph on which they apply, called host graph). In ATOM³, a transformation can be from a meta-model to another. But, as we represented every class (from UsiXML) we had to use in a single meta-model, we didn't use this possibility.

As in other graph-based transformation tools, the implementation of the meta-models, models and transformation rules is (almost) strictly declarative. But a few things can be coded in python (www.python.org), as we will describe further.

The only things that are obligatorily "coded" in ATOM³ are pre- and post-conditions, as well as some algorithms to give new variables a name based on other variables or change the name of existing variables, for example (in ATOM³, we can also add constraints on objects, and action triggered by certain events). These are coded here in Python, with a very simple syntax.

In the next section, we present the meta-model as we implemented it in ATOM³, and in the section after, we present all the transformations we implemented, each time with the precondition, postcondition, and the transformation itself.

3.2 Meta-model

As we can see on the figure 3-2, the meta-model is presented as an entity-relationship schema. The classes are represented as squares with the name above, and attributes under, and the relationships are represented by rhombuses. The cardinalities can also be implemented, and are here represented as sorts of attributes of the classes and relationships.

To create a meta-model in AToM³, we first have to choose in which formalism we want to create it. This formalism will be our “meta-meta-model”. And we have of course here to choose the entity-relationship model.

Now we can create our “meta-model”, with classes and relationships. For each class of relationship, we can (by double-clicking on it) create attributes, constraints, actions and a graphical representation.

Constraints are properties of the object that are checked by AToM³ at some specific moments (instantiation, connection, etc.). Actions are, as we can expect, actions that will be performed by AToM³ at specific moments (same possibilities as for constraints). Finally, the graphical representation will allow recognizing objects only with their appearing.

Inheritance is supported by AToM³, but not composition. To tackle with this limitation, we had to create bidirectional relationships instead of each composition relation. This is less readable and practical, but allows us to have the same result.

Another problem in AToM³ is the representation of the model itself. To create the model, we have a canvas of a limited size, and the objects we create in it are quite big. What’s more, the representation of the cardinalities as attributes of the objects makes it even bigger. So we have soon a not very clear schema, with arrows crossing objects, and this is not very readable and forces us to lose much time reorganizing the schema to have something clear.

Figure 3-2 is the entire meta-model implemented in AToM³. As the graph of figure 3-2 is too small to be readable, the figure 3-3 takes only the “task model” part of the meta-model as an example.

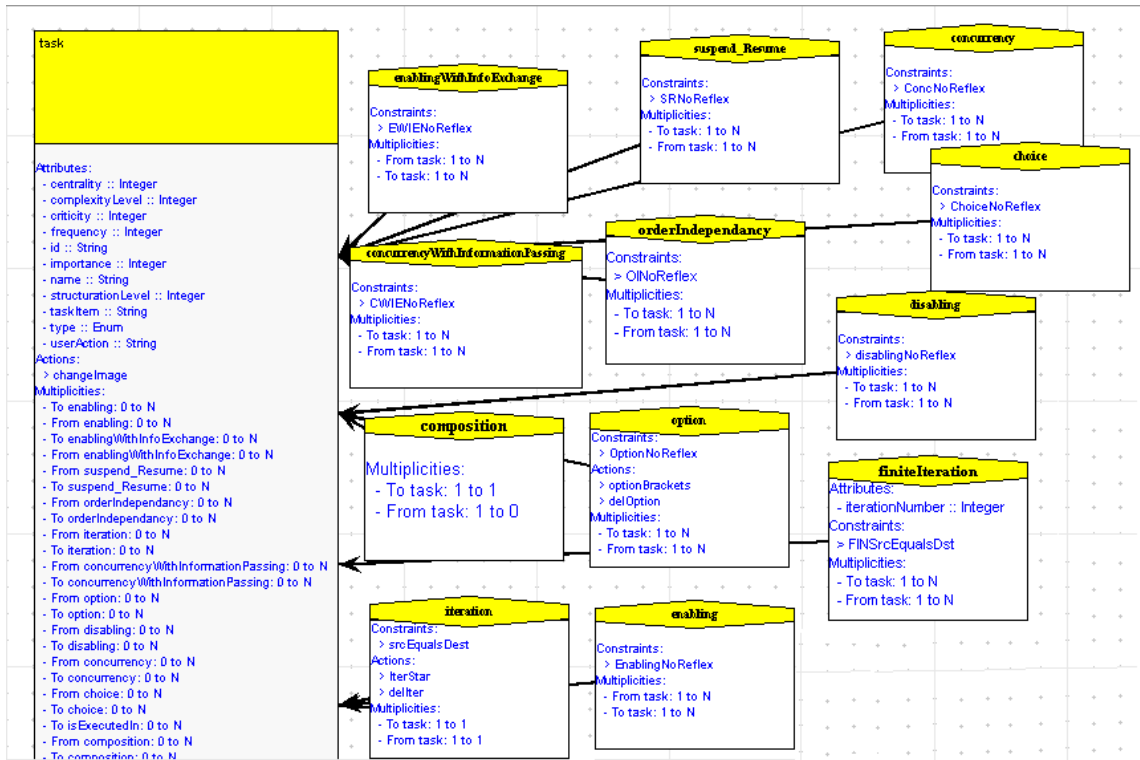


Figure 3-3. Task Model in ATOM³

3.3 Transformations

When we have finished creating our meta-model, we can generate buttons in ATOM³ to instantiate the objects of this meta-model. The objects will be instantiated with the graphical representation we gave it, or a default one. With this meta-model generated, we can now create the transformation rules.

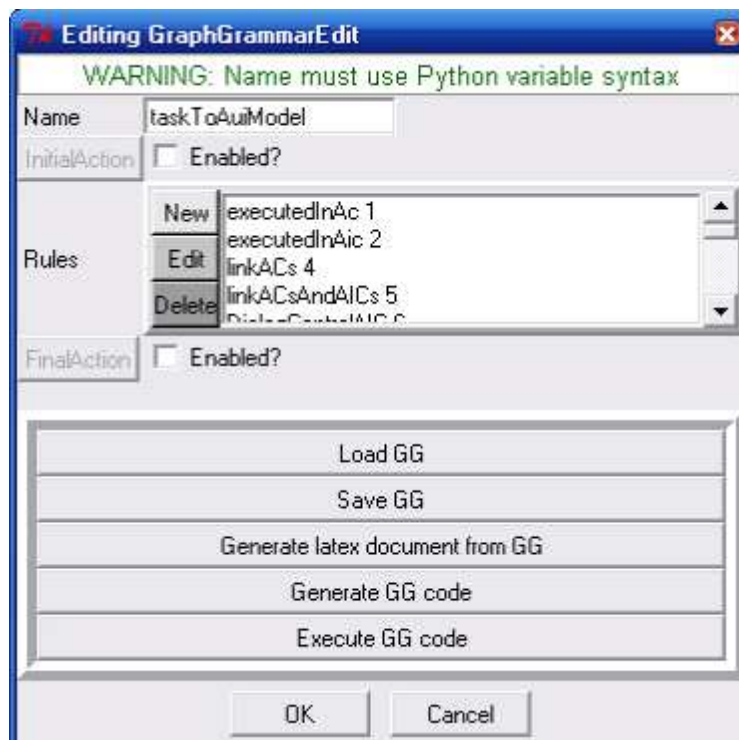


Figure 3-4. Rules set

3.3.1 The rules sets

In AToM³, creating rules sets is very simple. As we can see in the figure 3-4, there is a field to give the rule set a name, and a list in which we can very simply add (or remove) rules.

So it is very simple to create distinct rules sets in AToM³, and to save them. Re-loading them is very easy too.

When we have created our rule set, clicking on “Generate GG code” will generate python code from our rules. This code is the code that will be executed for our model transformations.

3.3.2 The rules

Here we show how the rules are implemented in AToM³. For a complete listing of all the rules we implemented in AToM³, see the appendix A.

Transformation rules are mainly composed of six components:

- The Left Hand Side (LHS)
- The Right Hand Side (RHS)
- The condition
- The action
- The order of the rule
- The name of the rule

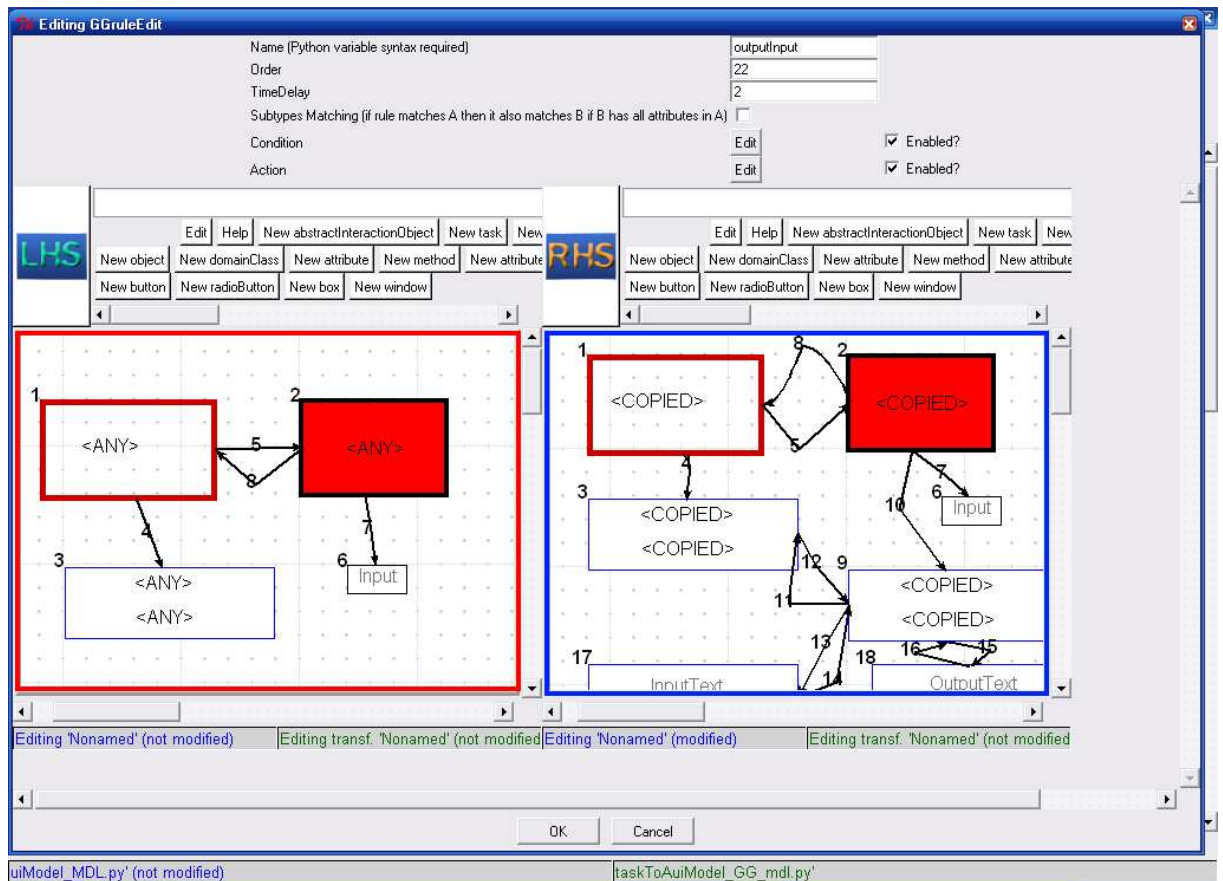


Figure 3-5. Transformation rules

There is no NAC in ATOM³, since it is replaced by a condition in python code. If this condition returns true, then the rule is executed.

The action is also a python code block, which is executed after the rule. The fact that, in ATOM³, condition and action are python code is very useful: while being a little harder to use and understand for an interface designer that has no (or few) programming skills, it allows things that couldn't be done graphically. In fact, everything could be done with only the action and condition, since ATOM³ allows implementing programmatically everything it can make graphically. But this is not a good idea, since code is harder to implement than graph and so, more subject to errors.

However, condition and action also allows (and that's their real purpose, in addition to the NAC role of the condition) determining values by browsing the model (indifferently source or target model, since there is no distinction) and computing the values from it. The condition can be constructed exactly like a NAC, but the pair condition/action can use a simple mechanism to ensure the rule is executed only one on a given object: the action adds the object an attribute and if the condition sees it, it means that the rule has already been executed, and the execution of this rule is stopped.

Finally, ATOM³ allows determining a fixed total order on the rules, but this order cannot be dynamically modified. There is no explicit flow control; the condition only allows preventing a rule to execute but, in that case, the rule will never be executed on the same pattern of the source model.

On figure 3-6, we can see a model transformation that corresponds to the following schema:

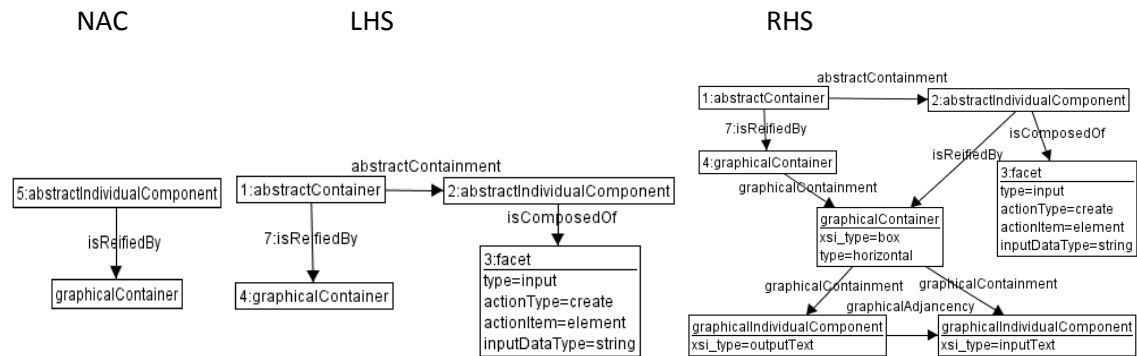


Figure 3-6. Graphical rule description

The main fields are name, and order, to give the rule an order in the rules set. There is also a checkbox: “subtype matching”: by checking it, we allow ATOM³ to recognize inheritance in the transformations, so for example if a box is a subtype of “GraphicalContainer”, it can be recognized as a graphical container in the LHS.

Then we come to the rule itself. It is divided into two parts: Left Hand Side (LHS, framed in red) and Right Hand Side (RHS, framed in red). The LHS is a pattern that will be searched by ATOM³ in the graph we want to transform. When ATOM³ finds it, it replaces it by the RHS.

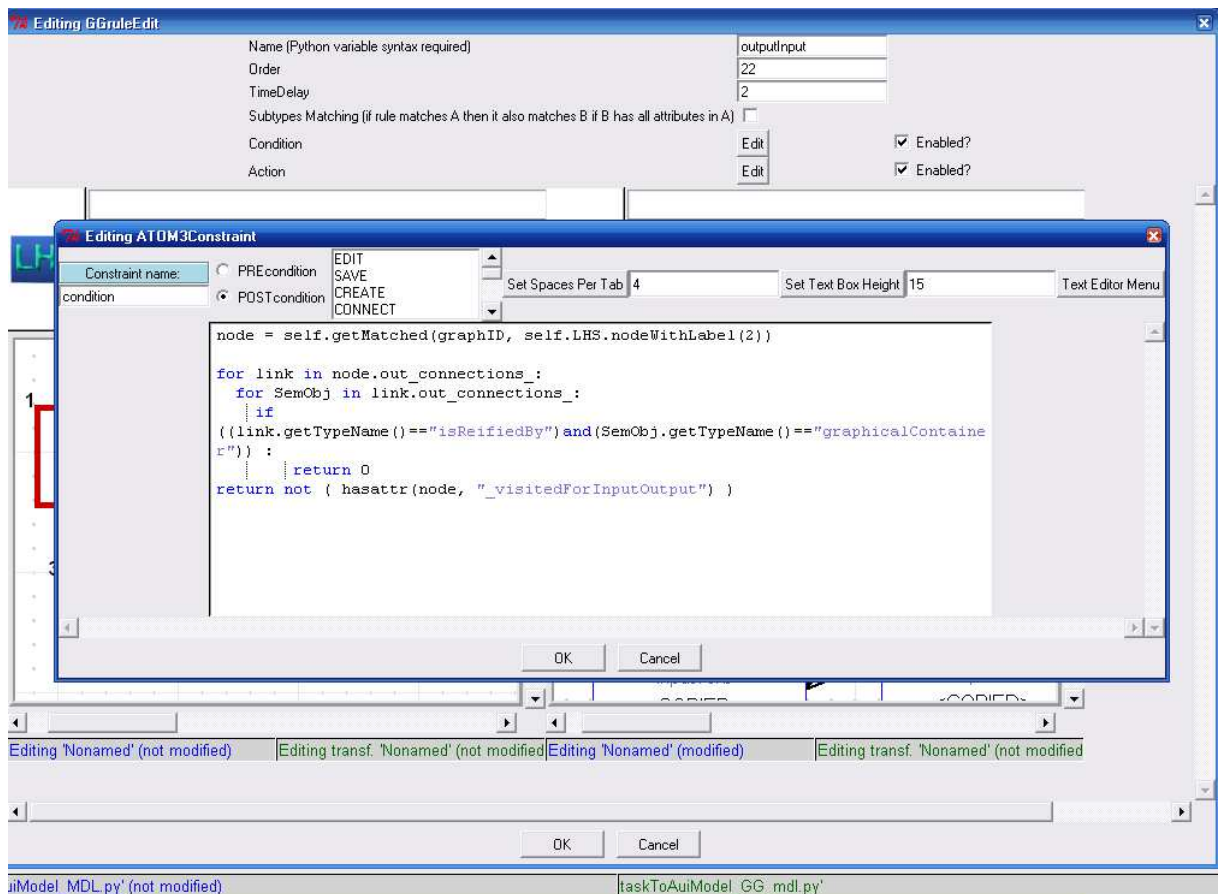


Figure 3-7. Condition

The (pre)condition of figure 3-7 corresponds to the NAC of figure 3-6. The first line,

```
node = self.getMatched(graphID, self.LHS.nodeWithLabel(2))
```

simply takes the object in the LHS that has the label 2 : it is the *AbstractIndividualComponent*.

```
for link in node.out_connections_:
```

means that we explore every link starting from the object. And

```
for SemObj in link.out_connections_:
```

means that we explore every object that is a target of this link. We do this to find if there is already a link between the *AbstractIndividualComponent* and a *GraphicalContainer*, which is well the NAC (precondition) of our transformation. If we find one, the precondition is not met, and we return 0, which means that the rule will not be executed.

The second part of the precondition is used to prevent the rule to be executed twice on the same objects. When the rule is executed on the object, we give it an attribute “_visitedForRule” (cf. action, figure 3-8): this allows ATOM³ to see it hasn’t to execute the rule once more.

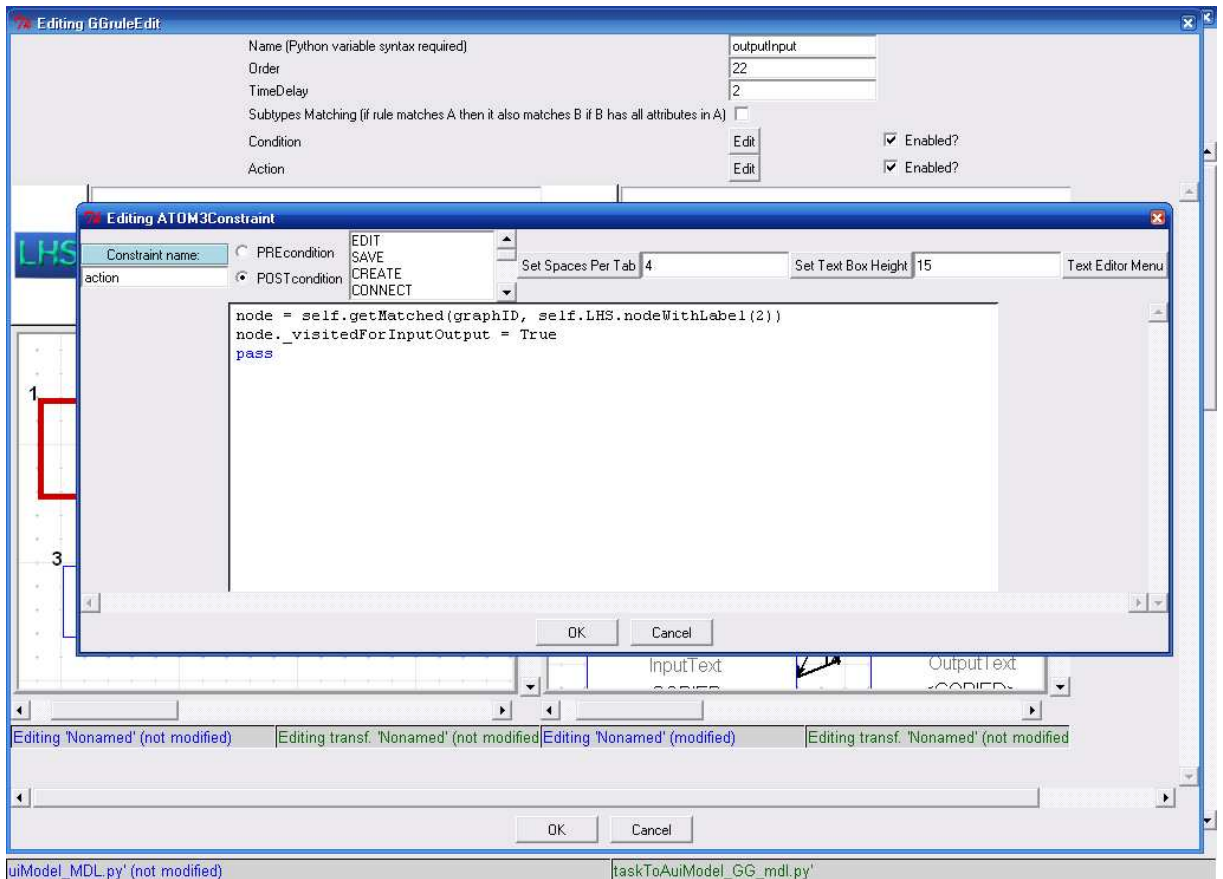


Figure 3-8. Action

The rule of the action here is to give the node an attribute, which will allow ATOM³ to see later that it has already executed the rule on this node.

3.3.3 Properties of the objects

ATOM³ also allows editing the properties of the objects in the RHS. Once more, this is very simple.

Figure 3-9 shows how we edit the properties of the object with label 11 in the RHS. As we can see in the edition window there are three possibilities to give a variable a value: “copy from LHS”, “Specify code”, or giving directly a value in the field next to the variable’s name.

Copy from LHS means that the variable will simply have the same value than it the object with the same label in the LHS. *Specify code* will allow us to create a small algorithm to determine the variable’s value.

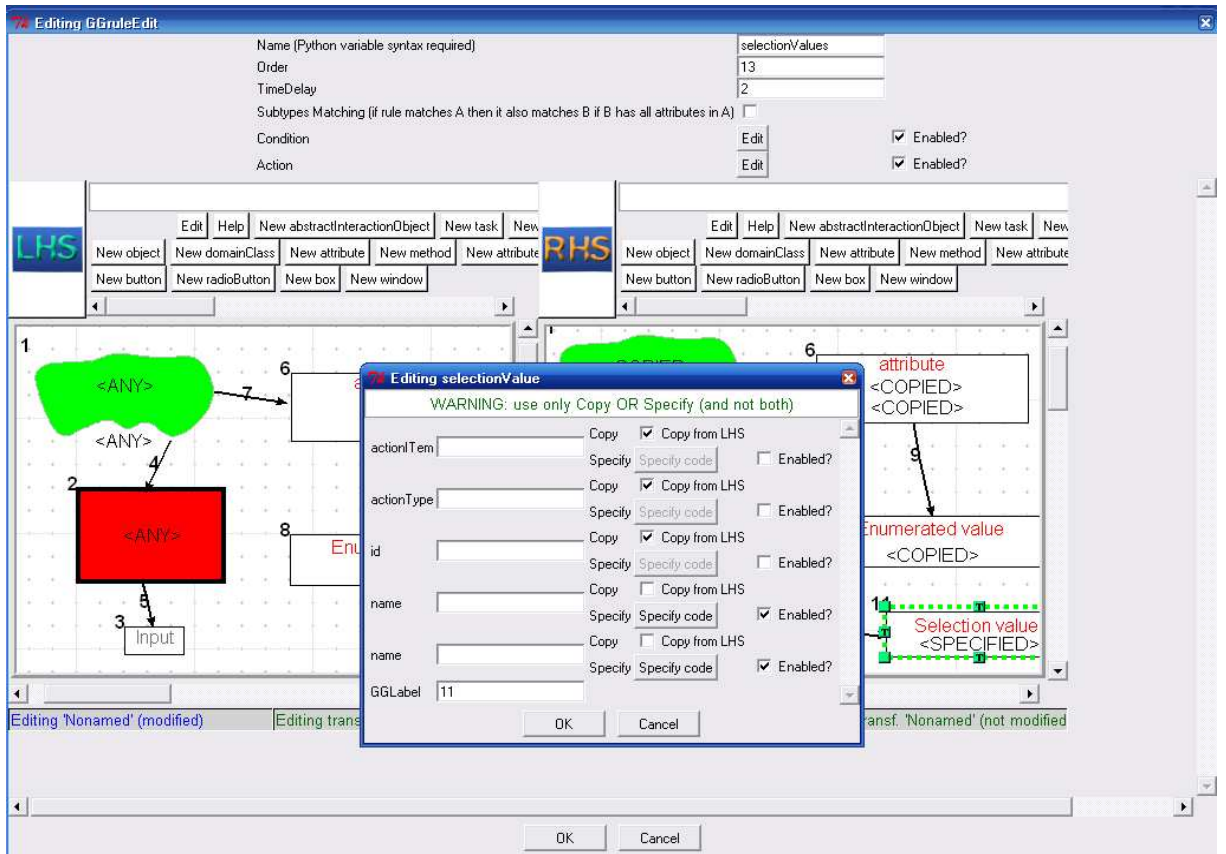


Figure 3-9. Object properties in transformation rule

In figure 3-10, we see the small algorithm that we used here to give the variable a value. In fact, it takes the name of the object will label 8 in the LHS as value. The syntax for such algorithms in ATOM³ is still very simple.

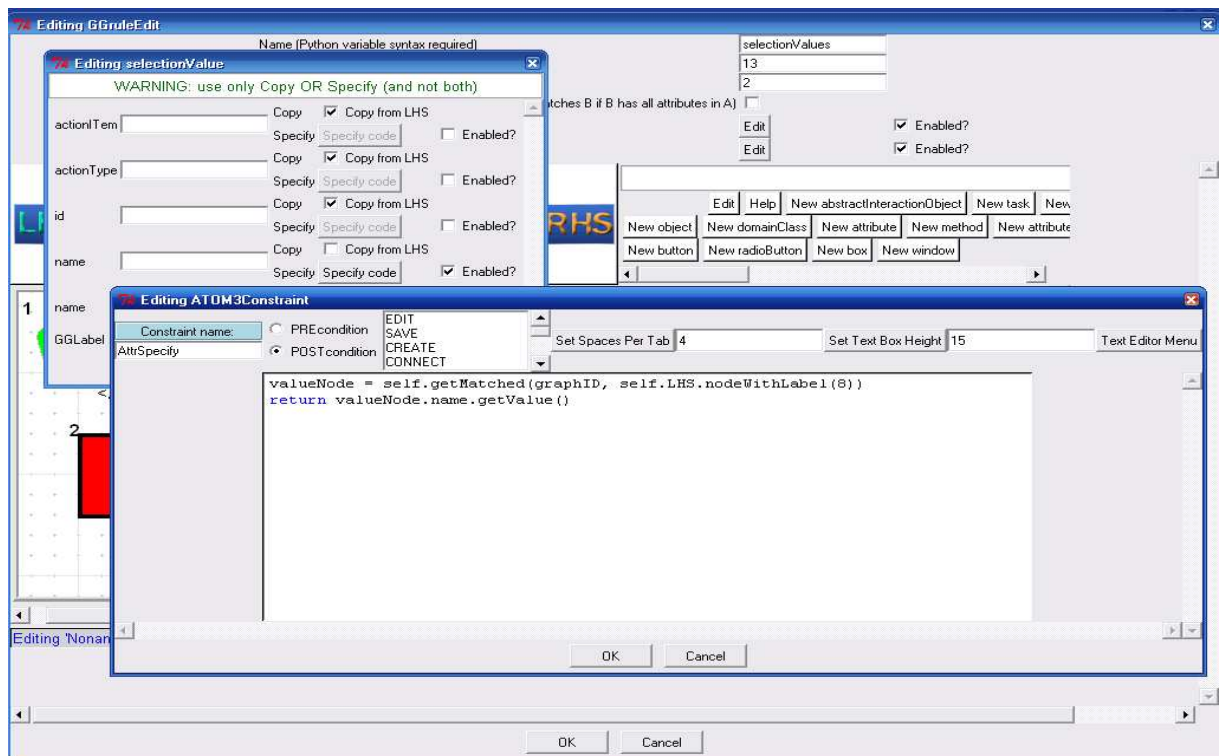


Figure 3-10. Value computing in transformation rule

Sometimes, the fact that we can, in ATOM³, use Python for pre-and post-conditions help us create simpler transformation rules.

Let us take for example the rule 8. The role of this task is : if two tasks are **[linked]** to, respectively an abstract container and an abstract individual component and if these two tasks are linked together by a dialog control relation, then the same dialog control relation must be created to link the abstract container and the abstract individual component.

But in the graph of the LHS, there is apparently no link between the task with label 1 and the task with label 2. We created it so because there can be several types of dialog control relations between two tasks. So, by putting it in the graph, we would have had to create a transformation rule for each type of dialog control relation.

Instead, we used the precondition to verify the relation. The execution of the rule is then as follows:

- ATOM³ finds the LHS pattern in the source graph.
- It then executes the precondition. This one says: if there is a link between the two tasks, and if this link is no of type “composition”, then add an attribute “link type” to the first task, and continue the transformation rule.
- ATOM³ now replaces the LHS by the RHS, and so creates a link between the abstract container and the abstract individual component.

- Finally, the symbol of the new dialog control relation is set by another python algorithm, that takes the value of the attribute “linkType” the precondition had added to the first task.

So, with only one transformation rule, all types of tasks relationships are processed.

This way of working is however a little “artificial”: instead of really creating a graph transformation rule in a declarative manner, we use python to make it more “polyvalent”. This means that the rule was harder to create, and doesn’t really respect the point of view of ATOM³, that should be (almost) strictly declarative. However, the earning in terms of time is here worth the implementation complexity.

Rule 8 (Order 8): DialogControlAICAC

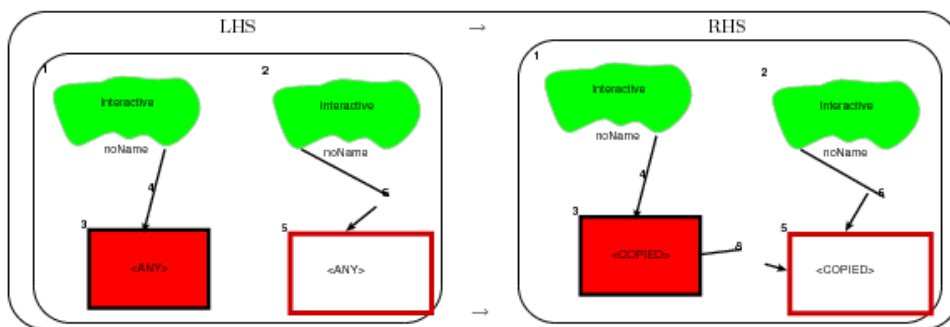


Figure 8: DialogControlAICAC

Precondition:

```
node1 = self.getMatched(graphID, self.LHS.nodeWithLabel(1))
node2 = self.getMatched(graphID, self.LHS.nodeWithLabel(2))
for link in node1.out_connections_:
    for SemObj in link.out_connections_:
        if (SemObj==node2) and (node.getTypeName()!='composition'):
            node1.linkType = link.getTypeName();
            return not ( hasattr(node2, '_visitedForDialogControl') )
return 0
```

Post action:

```
node = self.getMatched(graphID, self.LHS.nodeWithLabel(2))
node._visitedForDialogControl = True
pass
```

Specify symbol:String in *aviDialogControl #8*

```
node1 = self.getMatched(graphID, self.LHS.nodeWithLabel(1))
return node1.linkType
```

Figure 3-11: the rule in ATOM³ to derive abstract dialog control relationships

Finally, to help the reader understanding the models we implement in AToM³, we give in figure 3-12 the graphical representation of all the objects that will be met in the examples.

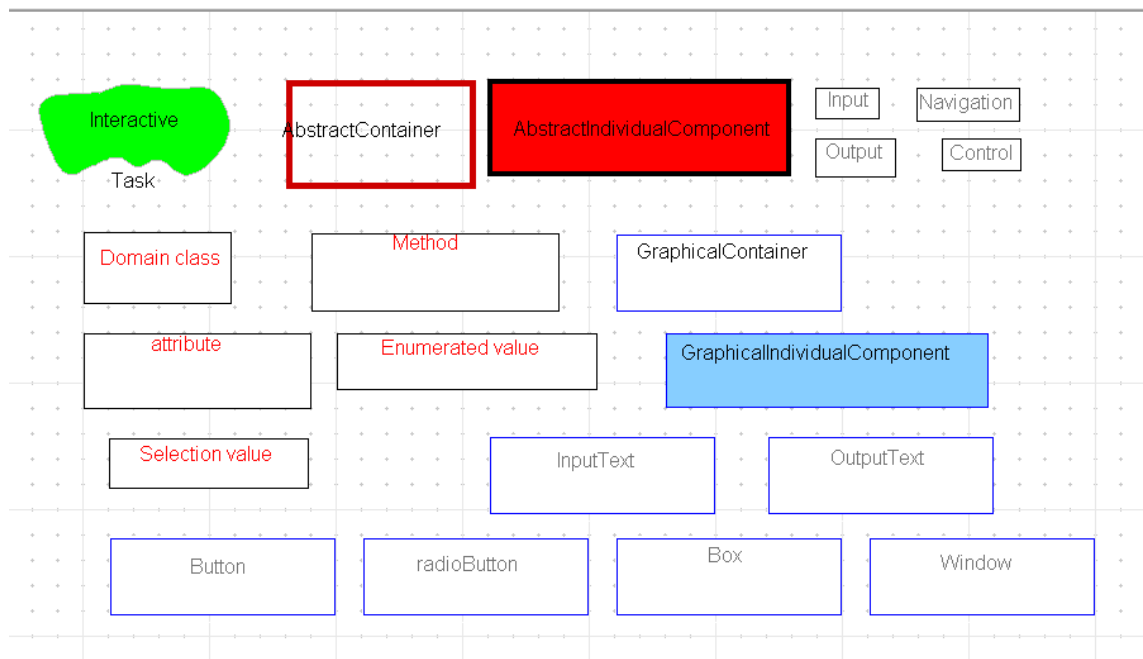


Figure 3-12. AToM³ Objects graphical representation

Chapter Four. Custom transformation engine in java

Instead of using an existing tool, as AToM³, the second possibility was to create a project entirely coded in the Java programming language. The goal here was to have a program that could take a UsiXML file as input, apply transformations on it, and give the result as an output, still in UsiXML format.

4.1 Implementation

The program itself is composed of five main classes, plus the classes created by jaxb. These four classes are:

- The GUI class
- The main class, which reads the UsiXML file, translates it in java objects, and instantiates the rules class. After all the rules are executed, the main class recreates a UsiXML file with the result.
- The rules class, which contains the methods of all the transformation rules
- The rulesHelpers class, which contains methods to help create transformation rules, for example "*findAC(Iterator auilter, string id)*" that allows to find an abstract container of id *id* with an iterator. This allows the rules in the rules class to be slightly shorter, and avoids a part of the redundancy.
- The rulesTree class, which contains a tree of all the transformation rules

When the GUI designer wants to implement new transformation rules, he has to modify the last two classes. Figure 4-1 and 4-2 are respectively the UML diagram of our transformation engine, and the sequence diagram of the execution of transformation rules in our transformation engine.

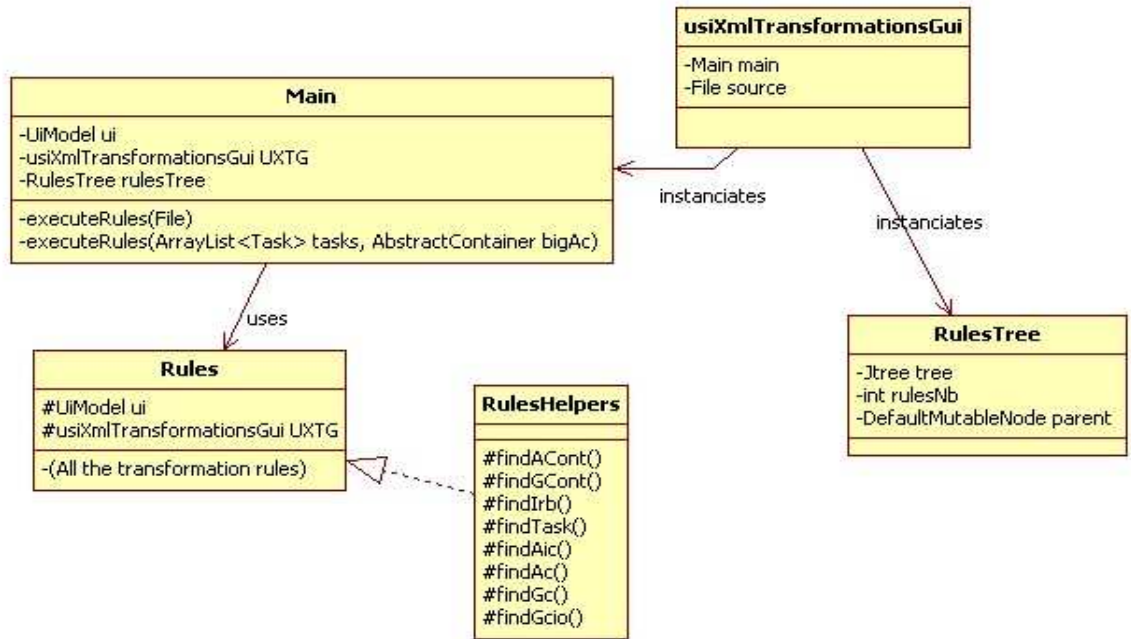


Figure 4-1. UML schema of the application

When executing transformations rules, the sequence is like below:

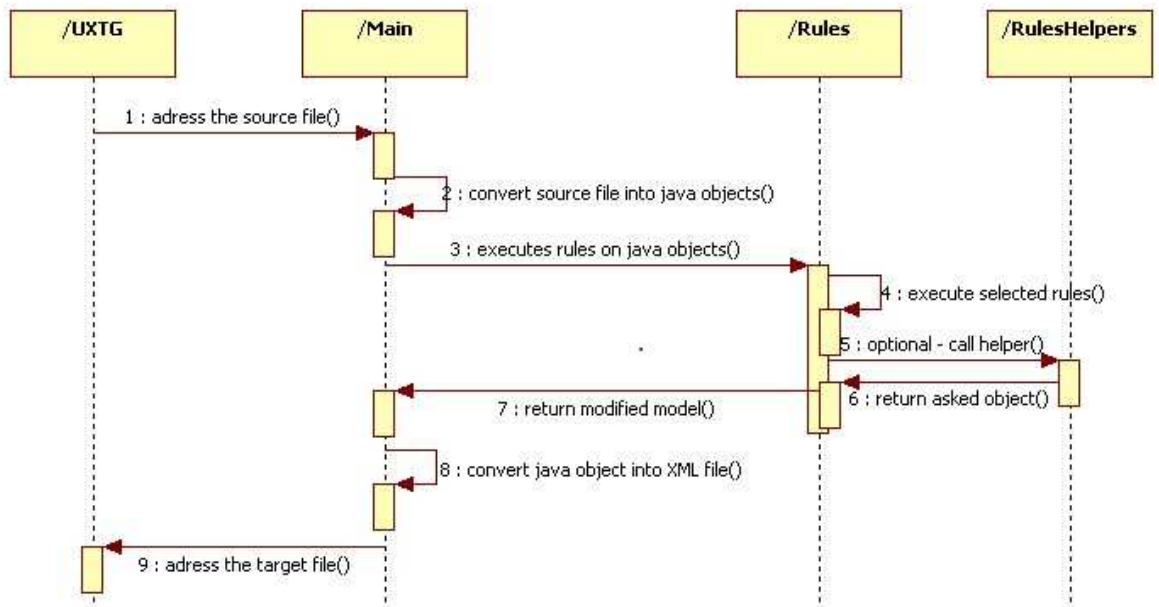


Figure 4-2. Sequence diagram of the application

First, using the GUI, we open the source file. When asking to execute rules, the interface instantiates Main, that will marshal the file (convert it into java objects). Then, the main instance will instantiate the Rules class and ask executeRules() to executes all the transformations rules. The Rules class, for some transformation rules, instantiates the RulesHelpers class to use one of its helper rules. Finally, the main class re-converts the java objects into an XML file, that is displayed on the graphical user interface.

All the classes have access to the model, which is directly modified. The source model then becomes at the end the target model; there is no distinction between them. All classes also have access to the graphical user interface, to be able to know which transformations rules are selected.

In the next chapter, we explain each class in detail.

4.2 Details of the implementation

Here, we explain the way each class is implemented, but we don't show the code. This one is given in appendix.

4.2.1 The graphical user interface (UsiXMLTransformationGui)

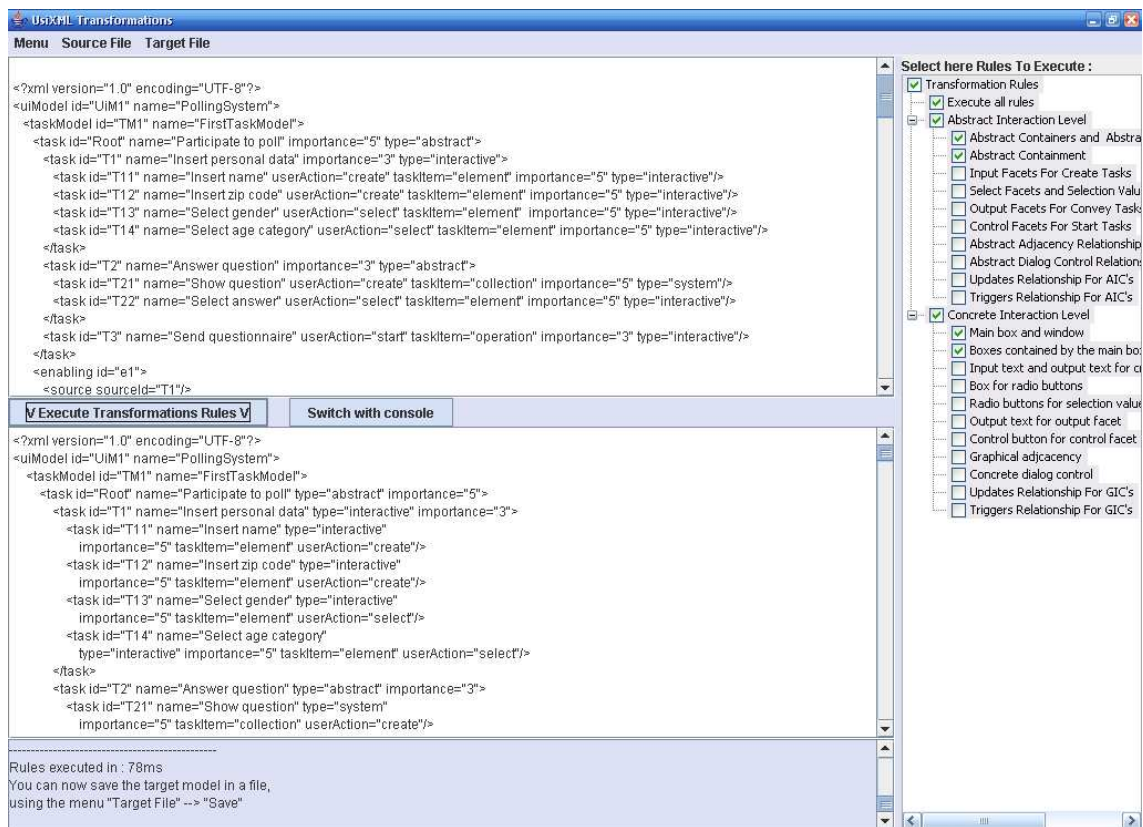


Figure 4-3. Graphical User Interface

The graphical user interface of our program has been made using swing. It is very simple, as we can see on the figure. Mainly, it has two big text areas that are used to show the source and target files:

- Source file is the UsiXML file before the execution of the transformation rules
- Target file is the UsiXML file obtained after applying the transformation rules on the source file

At the right side of the interface, we can see a tree containing all the implemented transformation rules, each with a checkbox aside. The checkboxes allow deciding whether or not the rule will be executed.

Between the two big text areas, are two buttons: "Execute rules" and "switch with console".

- “Execute rules” will execute all the transformation rules that have their checkboxes checked, or all of them if the checkbox of “All transformation rules” above is checked.
- Switch with console will simply take the text of the console and put it in the in-low text area, and vice-versa. It is just a small convenience for reading easily the text of the console.

Finally, important messages, such as errors, executed rules and time to execute them are displayed in the console.

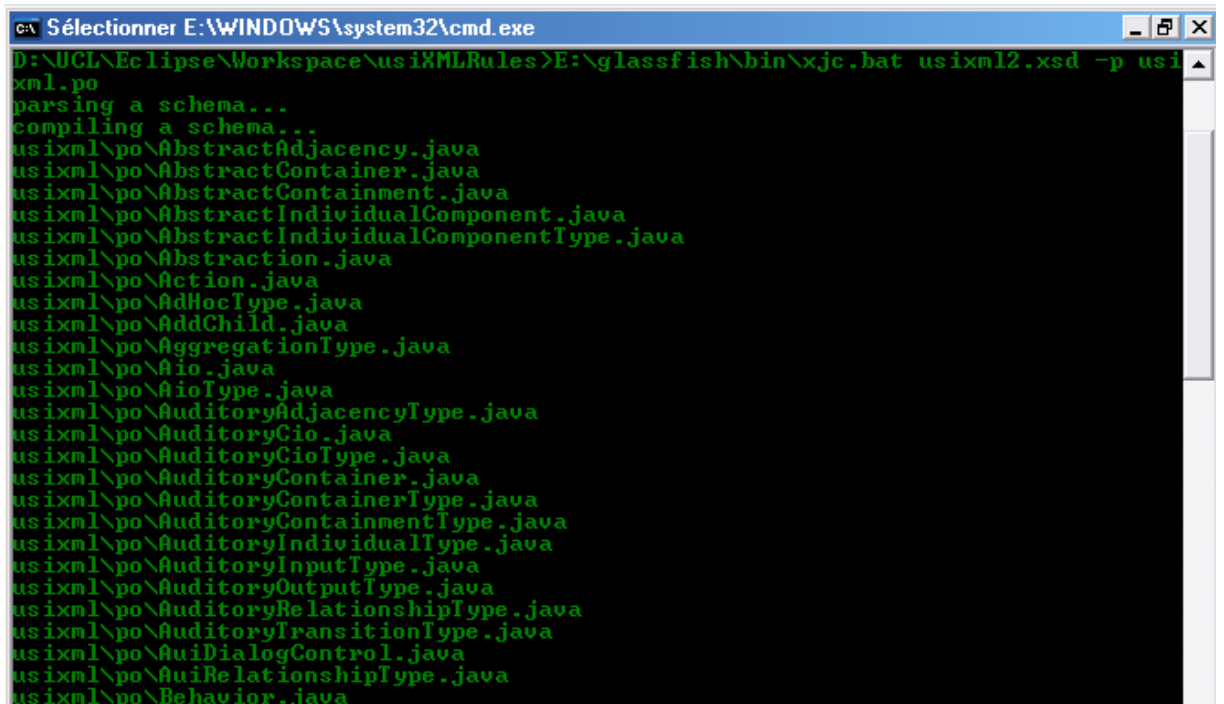
The interface has three menus: the first has only one element: “quit”, to exit the application. The second is used to open the source file, or save it if we made changes in it. Finally, the third is used to save the target file after the execution of the rules.

4.2.2 The main class

Many things are done in this class. It has to:

- Read the source UsiXML file and translate it into java objects.
- Instantiate the rules class, and call each rule.
- Create the target UsiXML file with the result.

First, we used `JAXB` to automatically create java classes corresponding to all entities in UsiXML³. JAXB can, with a schema in an *.xsd file as input, create java classes for all entities present in the schema. These classes have *getters* and *setters* for all attributes, and can thus be easily used as is (in fact, because of a compatibility problem with some type of lists, some attributes were missing, and we had to add them manually).



```
cmd Sélectionner E:\WINDOWS\system32\cmd.exe
D:\UCL\Eclipse\Workspace\usixmlRules>E:\glassfish\bin\xjc.bat usixml2.xsd -p usi
xml.po
parsing a schema...
compiling a schema...
usixml\po\AbstractAdjacency.java
usixml\po\AbstractContainer.java
usixml\po\AbstractContainment.java
usixml\po\AbstractIndividualComponent.java
usixml\po\AbstractIndividualComponentType.java
usixml\po\Abstraction.java
usixml\po\Action.java
usixml\po\AdHocType.java
usixml\po\AddChild.java
usixml\po\AggregationType.java
usixml\po\Aio.java
usixml\po\AioType.java
usixml\po\AuditoryAdjacencyType.java
usixml\po\AuditoryCio.java
usixml\po\AuditoryCioType.java
usixml\po\AuditoryContainer.java
usixml\po\AuditoryContainerType.java
usixml\po\AuditoryContainmentType.java
usixml\po\AuditoryIndividualType.java
usixml\po\AuditoryInputType.java
usixml\po\AuditoryOutputType.java
usixml\po\AuditoryRelationshipType.java
usixml\po\AuditoryTransitionType.java
usixml\po\AuiDialogControl.java
usixml\po\AuiRelationshipType.java
usixml\po\Behavior.java
```

Figure 4-4. using JAXB

The program reads the UsiXML file (*.usi), and instantiates the classes (created by JAXB) to internally represent the file (thus, using java objects), using the Castor project.

³ Cf. Figure 1 p. 5

```

// mappingUsiXml.xml contains rules to translate XML objects into
// java objects, and vice-versa.
Mapping map = new Mapping();
map.loadMapping("mappingUsiXml.xml");
Unmarshaller h_unmarshaller = new Unmarshaller(map);
// The file is translated into java objects.
ui = (UiModel)h_unmarshaller.unmarshal(in);

```

Figure 4-5. Reading and translating a UsiXML file

As we see in the code, Castor needs a *mapping file*⁴ to correctly read the UsiXML file (and create the new file after the transformation rules have been applied). A mapping file is a file that contains rules (expressed in XML) to create an XML file from java objects. For example, the code in the mapping file for the *AbstractContainer* object is:

```

<class name="usixml.po.AbstractContainer" auto-complete="false">
  <description>Default mapping for class usixml.po.AbstractContainer</description>
  <map-to xml="abstractContainer"/>
  <field name="orderType" type="string" required="false"
    direct="false" transient="false">
    <bind-xml name="order-type" node="attribute" reference="false"/>
  </field>
  <field name="name" type="string" required="false" direct="false" transient="false">
    <bind-xml name="name" node="attribute" reference="false"/>
  </field>
  <field name="splittability" type="boolean" required="false"
    direct="false" transient="false">
    <bind-xml name="splittability" node="attribute" reference="false"/>
  </field>
  <field name="id" type="string" required="false" direct="false" transient="false">
    <bind-xml name="id" node="attribute" reference="false"/>
  </field>
  <field name="abstractContainerOrAbstractIndividualComponentOrInput" type="usixml.po.AioType"
    collection="arraylist" required="false" direct="false" transient="false">
    <bind-xml node="element" reference="false" auto-naming="deriveByClass"/>
  </field>
</class>

```

We can see in the code that it corresponds to the class "usixml.po.AbstractContainer". It will be mapped in the new XML file with the name "abstractContainer". Then we see each attribute of the class AbstractContainer, and for each one, we see the name of it in the new XML file, as well as how it will be formatted (attribute or node).

The last attribute is a little special: this attribute is a list. And, at the end of the "bind-xml" element, we see the attribute "auto-naming="deriveByClass". This means that the name of the attribute (in the new xml file) will be dynamically chosen by Castor, following the name of the class. As an AioType can be an instance of AbstractContainer as well as AbstractIndividualComponent or Input (all extend AioType), the name can be one of the three.

4

4.2.3 The Rules class

In this class, we find all the methods for the transformation rules. To make the program easier to read and modify, there is one method per transformation rule. Of course, this means a small loss of 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. But it was the only way to allow the maintainability of the code.

The main difficulty in the java programming method for the creation of transformation rules is that Java isn't designed for pattern matching. That forbids us to take a transformation rule expressed in UsiXML to automatically execute it by searching its LHS in the UsiXML file and replacing it with the RHS. Each transformation rule has to be coded "by hand".

Here, we see an example of such a transformation rule expressed in Java:

```
void abstractAdjacency(){
    /*
     * For each couple of sister tasks executed into AIOs,
     * we define an abstractAdjacency relationship between these AIOs
     */
    this.UXTG.console.append("Abstract Adjacency...");
    Iterator relIter = ui.getTaskModel()
        .getTaskRelationshipOrDecompositionOrTemporal().iterator();
    while (relIter.hasNext()){
        // Searching through the list of TaskRelationship's,
        TaskRelationshipType rel = (TaskRelationshipType)relIter.next();
        // we try to find a relation of type "EnablingType"
        if (rel.getClass()==(new EnablingType()).getClass()){
            Iterator sourceIter = rel.getSource().iterator();
            // and we create a abstractAdjacency relation for every tasks
            // that are linked by an enabling relation.
            while (sourceIter.hasNext()){
                Source src = (Source)sourceIter.next();
                Iterator targetIter = rel.getTarget().iterator();
                while (targetIter.hasNext()){
                    Target tar = (Target)targetIter.next();
                    if (!(src.getSourceId().equals(tar.getTargetId()))){
                        AbstractAdjacency AA = new AbstractAdjacency();
                        AA.setId("AA" + rel.getId());
                        AA.getSource().add(src);
                        AA.getTarget().add(tar);
                        ui.getAuiModel()
                            .getAuiRelationshipOrAbstractContainmentOrAbstractAdjacency()
                            .add(AA);
                    }
                }
            }
        }
    }
}
```

Figure 4-6. Abstract Adjacency Rule

In the figure 4-6, we see the rule. The goal of this one is to create a relation of type "AbstractAdjacency" for each couple of sister tasks executed into AIOs that will thus be next to each other in the Final User Interface.

Expressing transformation rules in java is much more complex than in AToM³ or ATL. To show how these rules work, we will show here three of them, in ascending order of complexity.

First, let us describe a simple rule:

```
void abstractContainment(Iterator taskIter){

/* For each abstract container, an abstract containment relationship
 * is created, with this abstract container as source, and, as
 * targets, all the abstract individual component corresponding to the
 * tasks that compose the task corresponding to the abstract ontainer.
 */

while (taskIter.hasNext()){
//
    Task t = (Task)taskIter.next();
    String id = t.getId();
    if (!(t.getTask().isEmpty())){
        // Tasks is composed of subtasks.
        abstractContainment(t.getTask().iterator());
    }

    Iterator subTasksIterator = t.getTask().iterator();
    // We browse all the subtasks composing the task t
    while (subTasksIterator.hasNext()){
        Task subT = (Task)subTasksIterator.next();
        Target tar = new Target();
        if (subT.getTask().isEmpty())
            // If the subtask has no subtasks,
            // it is an Abstract Individual Component,
            // so we give it "AIC_" plus the task id as id.
            tar.setTargetId("AIC_" + subT.getId());
        else
            // If the subtask has subtasks,
            // it is an Abstract Container,
            // so we give it "AC_" plus the task id as id.
            tar.setTargetId("AC_" + subT.getId());
        AbstractContainment aCont = new RulesHelpers(ui,
            UXTG).findACont(ui.getAuiModel().
            getAuiRelationshipOrAbstractContainmentOrAbstractAdjacency().
            iterator(),"AC_"+id);
            // We search for an already existing abstract
            // containment relationship with the abstract
            // container as source.
            if (aCont!=null){
                // If there is already a relationship, we simply add
                // the target to it.
                aCont.getTarget().add(tar);
            }
            else {
                // else, we create a new relationship with the
                // abstract container as source, and the
                // abstract individual component (or abstract
                // container) corresponding to the subtask
                // as target.
                AbstractContainment AbsCont = new
                    AbstractContainment();
                Source src = new Source();
                src.setSourceId("AC_" + id);
            }
    }
}
```

```

        AbsCont.getSource().add(src);
        AbsCont.getTarget().add(tar);
        AbsCont.setId("ACont_" + src.getSourceId());

        ui.getAuiModel()
        .getAuiRelationshipOrAbstractContainmentOrAbstractAdjacency()
        .add(AbsCont);
    }

}
    this.AbstractContainment=true;
}
}
}

```

This rule creates abstract containment relationships for abstract containers containing abstract individual components.

We first have to navigate all the tasks of the task model with an iterator to find ones that are composed of subtasks. For each of them, we create an abstract containment relationship with the abstract container corresponding to the task as source, and abstract containers of abstract individual components corresponding to the subtasks as targets. We recursively call the method on the subtasks that are also composed of subtasks.

The following code block:

```

if (subT.getTask().isEmpty())
    // If the subtask has no subtasks,
    // it is an Abstract Individual Component,
    // so we give it "AIC_" plus the task id as id.
    tar.setTargetId("AIC_" + subT.getId());
else
    // If the subtask has subtasks,
    // it is an Abstract Container,
    // so we give it "AC_" plus the task id as id.
    tar.setTargetId("AC_" + subT.getId());
    AbstractContainment aCont = new RulesHelpers(ui,
    UXTG).findACont(ui.getAuiModel().
    getAuiRelationshipOrAbstractContainmentOrAbstractAdjacency().
    iterator(),"AC_"+id);

```

is used to give an id to the abstract objects. The id's are always defined the same way:

- The abstract containers have "AC_" + the id of the task as id
- The abstract individual components have "AIC_" + the id of the task as id

This first rule itself is very simple but, as we can see, the implementation is already quite long.

For the comparison, the figure 4-7 shows the same rule implemented in AToM³:

Rule 1 (Order 1): `executedInAc`

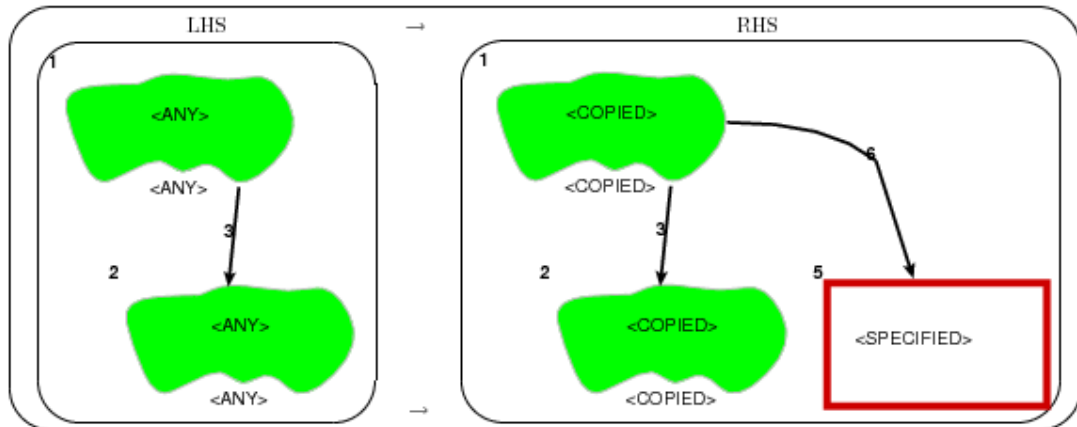


Figure 1: `executedInAc`

Precondition:

```
node = self.getMatched(graphID, self.LHS.nodeWithLabel(1))
return not ( hasattr(node, '_visitedForAC') )
```

Post action:

```
node = self.getMatched(graphID, self.LHS.nodeWithLabel(1))
node._visitedForAC = True
pass
```

Specify name:String in *abstractContainer* #5

```
return self.getMatched(graphID, self.LHS.nodeWithLabel(1)).name.getValue()
```

Figure 4-7. The rule in ATOM³

With more complex rules, things become worse.

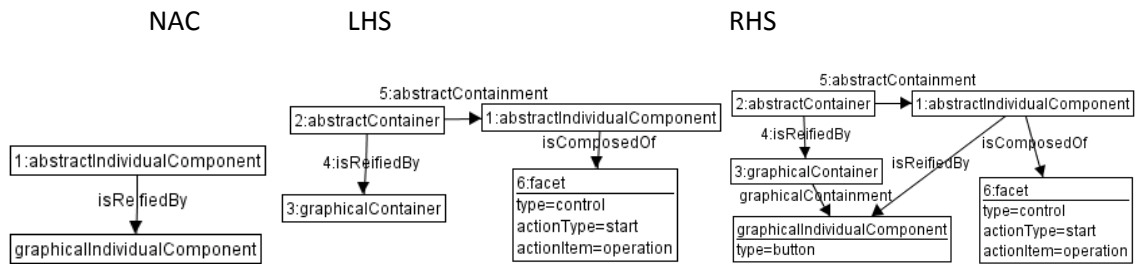


Figure 4-8. Generation of a control button

For each abstract individual component AIC composed of a control fact, and contained by an abstract container AC reified by graphical container GC, the following is done:

- A **Button** graphical individual component is created.
- A **graphical containment** relationship is created with GC as source, and the button as target.
- A **is reified by** relationship is created with AIC as source and the Button as target.

In AToM³, this rule is still quite simple because it doesn't involve many objects, and only one is created. But the method in java is very long, the figure 4-9 shows it:

```

void controlButtonForControlFacet() {

    UXTG.console.append
    ("Creating Control Button for Control Facet... \n");

    Iterator RelIter = ui.getAuiModel().
    getAuiRelationshipOrAbstractContainmentOrAbstractAdjacency()
    .iterator();
    Vector<IsReifiedBy> irbs = new Vector<IsReifiedBy>();

    // 1. We browse all the relations to find "abstract
    //   containment" relations
    while (RelIter.hasNext()) {

        Object rel = RelIter.next();
        if (rel.getClass().equals
            (new AbstractContainment().getClass())) {

            AbstractContainment contains =
                (AbstractContainment)rel;
            Source src = contains.getSource().get(0);
            // "AbstractContainment" relationships
            // always have only one source.
            Iterator mapIter =
                ui.getMappingModel().

```

```

getInterModelRelationshipOrManipulatesOrIsExecutedIn().
    iterator();
// 2.2 We search all the "is reified by"
//      relations with src as source
while (mapIter.hasNext()){

    Object map = mapIter.next();
    if ((map.getClass()).equals(new
        IsReifiedBy().getClass())){

        IsReifiedBy IR = (IsReifiedBy) map;

        Source IRSrc = IR.getSource().get(0);
        Target IRTar = IR.getTarget().get(0);
        if (IRSrc.getSourceId().equals(src.getSourceId())){
            // 2.4 We search a Graphical Container that has the
            // same id as the target.
            String id = IRTar.getTargetId();
            // 2.5 for that,we browse all the graphical
            //      containers
            Iterator cioIter =

ui.getCuiModel().getCioOrFinalComponentOrAuditoryCio().
    iterator();

        GraphicalContainerType gc =
            new RulesHelpers(ui, UXTG,
                rulesTree).findGC(cioIter, id);

        if (gc!=null){
            // We have a GraphicalContainer
            String cioId = id;

            // Ok, Src is reified by a graphical container
            // Now we browse all the targets, and if one is
            // composed of an input facet (create, string),
            // then we execute the rule.
            Iterator tarIter = contains.getTarget().iterator();
            while (tarIter.hasNext()){
                Target tar = (Target)tarIter.next();
                //Target tar = ((Target)(tarIter.next()));
                String tarId = ((Target)tar).getTargetId();

                // Now we browse all the AIO's to find a AIC
                // that is composed of an input facet (create)
                // and that has the ID of target.

                AbstractContainer ac = new RulesHelpers(ui,
                    UXTG, rulesTree).
findAC(ui.getAuiModel().getAioOrAbstractContainer().
        iterator(), tarId);
                AbstractIndividualComponent aic = null;
                if (ac==null){
                    aic = new RulesHelpers(ui, UXTG, rulesTree).
findAIC(ui.getAuiModel().getAioOrAbstractContainer().
        iterator(), tarId);
                    if (aic!=null){
                        controlButtonForControlFacetHelper
                            (src, cioId, aic, irbs);
                    }
                }
            }
        }
    }
}

```

```

Iterator irbIter = irbs.iterator();
while (irbIter.hasNext()){
    IsReifiedBy IRB = (IsReifiedBy)irbIter.next();
    ui.getMappingModel().

getInterModelRelationshipOrManipulatesOrIsExecutedIn().add(IRB);
    }
}

void controlButtonForControlFacetHelper
(Source src, String CioId, AbstractIndividualComponent aic,
Vector<IsReifiedBy> irbs){

    // We browse all the facets of the AIC to find an input
    // facet (create, string)
    Iterator FaIter = aic.getFacetOrInputOrOutput().iterator();
    //System.out.println("3.3. BOUTON !");
    System.out.println("AIC : " + aic.getName());
    while (FaIter.hasNext()){
        Object FA = FaIter.next();
        System.out.println("type : " + FA.getClass());
        if (FA.getClass().equals(new Control().getClass())){
            // We found an output
            Control control = (Control)FA;

            if ((control.getActionType().equals("start"))){
                // It is the type of input we were
                // looking for
                ButtonType button = new ButtonType();
                button.setId("Button_"+control.getId());

                IsReifiedBy irb = new IsReifiedBy();
                Source irbSource = new Source();
                irbSource.setSourceId(aic.getId());
                Target irbTarget = new Target();
                irbTarget.setTargetId(button.getId());
                irb.getSource().add(irbSource);
                irb.getTarget().add(irbTarget);

                // We cannot directly add it in the
                // mapping model because an iterator is
                // currently browsing it, so it create a
                // ConcurrentModificationException.
                // So we had it in a vector, that will
                // further be browsed to add all irbs
                // into the mapping model.
                irbs.add(irb);

                GraphicalContainmentType butGCT = new
                    GraphicalContainmentType();

                Source butSrc = new Source();
                butSrc.setSourceId(CioId);
                butGCT.getSource().add(butSrc);
                Target outTar = new Target();
                outTar.setTargetId(button.getId());
                butGCT.getTarget().add(outTar);

                ui.getCuiModel().getCioOrFinalComponentOrAuditoryCio().add(butto

```

```
n);  
                                ui.getCuiModel().  
    getCuiRelationshipOrCuiDialogControlOrGraphicalAdjacency().add(b  
utGCT);  
                                }  
    }  
}
```

Figure 4-9. A complex transformation rule in java

Obviously, maintaining such a code is a very difficult task. So, the maintainability and flexibility of our custom java transformation engine is very poor compared to the one of AToM³. Because of this complexity, implementing the whole rules set we decided to has been very long, around three man-months. That is more than two times more than AToM³ requires for the same task.

Chapter Five: ATL

First, let us remind that we didn't use ATL, so the description here is only intended to show how ATL executes the transformation rules to add to it the comparison.

ATL come under the form of an Eclipse (www.eclipse.org) plug-in. It is in fact composed of three things:

- The ATLAS transformation language
- The ATL execution engine
- ATL development tools (ADT)

ATL is a transformation tool for model driven engineering, implemented as an Eclipse plug-in and using its own transformation language. ATL follows a hybrid approach: transformation rules can be fully declarative, fully imperative or both declarative and imperative.

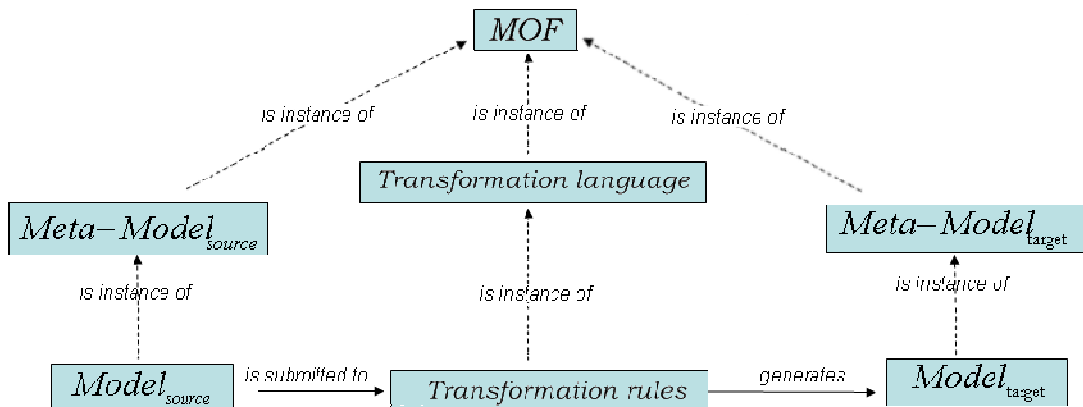


Figure 5-1 ATL execution engine

Each model conforms to a meta-model. The source model, transformation model and target model are all models conforming to their respective meta-models, which conform to the MOF recommendations.

The meta-model of the transformation is the ATLAS transformation language. It is both declarative, which allows simple and quite intuitive specification of transformation patterns, and imperative to help design complex rules which couldn't be with only declarative constructs.

The declarative rules are called “matched rules”. These are executed each time their source pattern is matched with a part of the source model. When the rule is executed, its target pattern is created in the target model.

Imperative rules are “called rules”. They are used in ATL to provide some mechanisms needed to create more complex rules.

Instead of a source pattern, like the matched rules have, imperative rules have parameters (and can be explicitly called). But called rules can have a target (like matched rules) instead of imperative code.

And declarative rules can also have imperative blocks, which can serve as entry or exit point.

Called rules allow explicit control of the execution, which declarative rules don’t (see the execution engine section).

ATL supports different model handlers that allow it to use different formats as sources and targets, such as XMI, XML documents, binary files and textual representations⁵ of models.

The ADT (ATL Development Tools) bring a certain number of facilities for the designer [ATL Desc.], such as syntax highlighting, error reporting and a debugger (Source-level debugging, Stepping through elementary operations and Breakpoints support). It also benefits from the use of the Eclipse IDE, with all the facilities this implies.

We didn’t implement our rules catalog in ATL, but ATL is one of the currently most used transformation engines, so we add to the comparison.

⁵ ATL can serialize models to text format (using the TCS model handler)

Chapter Six. Case studies

In this section, we will show how the transformation rules we implemented are applied on models. We will begin by two very simple examples and describe in detail the execution of the transformations, as well as specific issues for each tool.

Each example will be executed using AToM³ and our custom java transformation engine (we didn't implement the transformation rules in ATL, so we cannot show the ATL execution).

For each example, we first describe it in natural and graphical language, then show the source model used with each tool and finally, the result of the execution of the transformation rules.

The chapter will be structured as follows:

- First, we will describe the rules we implemented in AToM³ and our custom transformation engine, and why we implemented these ones.
- Then we will illustrate some rules by two simple examples.
- Finally, we will apply our rule set to a bigger example: the virtual polling system.

But first, we describe the transformation rules we implemented.

6.1 Implemented rules

The set of rules we implemented is the same for AToM³ and our java transformation engine. It is in fact originally taken out from [Limbourg], but the example has been a little developed in [Stanciulescu], so we based ourselves on both to create our rule set.

The whole rules catalog is described in Appendix B, and rules are explained in the examples below.

6.2 Use case: Currency convertor⁶

This use case is intended to show how the currency convertor that can be found at the site <http://www.xe.com/ucc/> could have been implemented using the MDA approach. The way this currency convertor works is very simple: the user enters an amount in a text field, chooses the source and target currency and ask the conversion.

Here is the task model designed with the tool idealXML:

The task model is graphically described using IdealXML tool. The figure 6-1 depicts a CTT representation of the task model of the system. The root task consists of converting currencies. The user has to provide the amount he wants to convert, as well the current and the wanted currencies. Then the user launch the conversion and finally, the system displays the result.

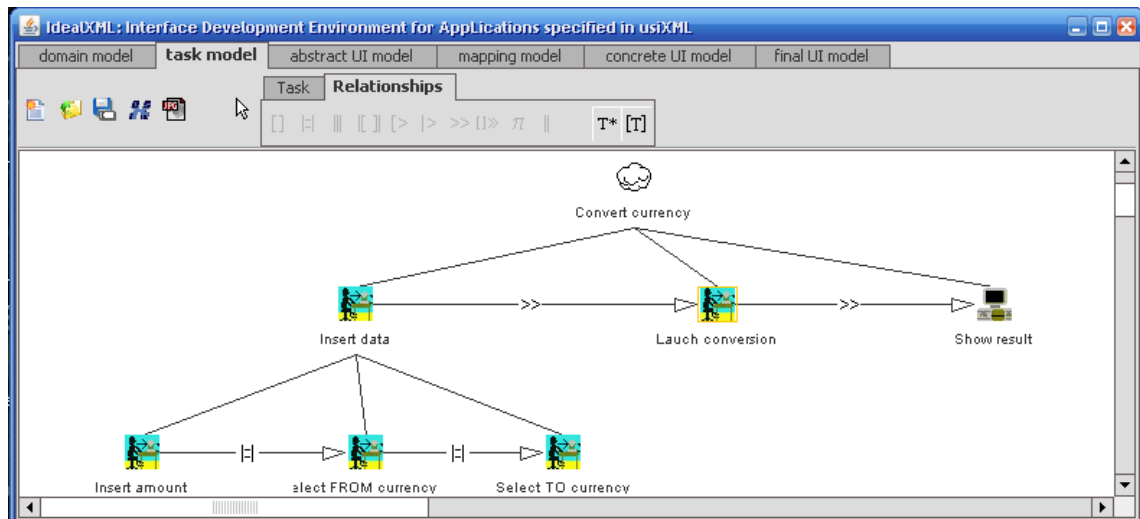


Figure 6-1. Currency convertor task model

The objective here is to have a simple example to show how the transformation rules work. So, we won't model the domain and mapping models.

⁶ The currency convertor can be found at the site : <http://www.xe.com/ucc/>

Here is the UsiXML specification generated by idealXML.:

```
<?xml version="1.0" encoding="UTF-8"?>
<uiModel>
  <taskModel id="tm0" name="taskmodel">
    <task id="Root" name="Convert currency" type="abstract">
      <task id="task1" name="Insert data"
        type="interaction">
        <task id="task2" name="Insert amount"
          type="interaction" />
        <task id="task3" name="Select FROM currency"
          type="interaction" />
        <task id="task4" name="Select TO currency"
          type="interaction" />
      </task>
      <task id="task6" name="Launch conversion"
        type="interaction" />
      <task id="task7" name="Show converted currency"
        type="application" />
    </task>
    <enabling id="e1">
      <source sourceId="task1" />
      <target targetId="task6" />
    </enabling>
    <enabling id="e2">
      <source sourceId="task6" />
      <target targetId="task7" />
    </enabling>
    <orderIndependence id="OI1">
      <source sourceId="task2" />
      <target targetId="task3" />
    </orderIndependence>
    <orderIndependence id="OI2">
      <source sourceId="task3" />
      <target targetId="task4" />
    </orderIndependence>
  </taskModel>
</uiModel>
```

Step 2: From Task a to AUI Model

The second transformation step involves a transformation system that contains rules applied in order to realize the transition from the task to the abstract model. This step is subdivided into five sub-steps according to [Limb04b]. We give here the set of rules that will be executed on this example (there are only a few, because here we only modeled the task model).

Sub-step 2.1: Rules for the identification of AUI structure

Rules 1 and 2 create abstract containers for tasks that have task children and abstract individual components for leaf tasks. Tasks 4 and 5 reconstruct the containment relationships for these AC's and AIC's.

The result of the application of these rules over the task model structure consists in a hierarchical decomposition of the AUI into abstract containers and abstract individual components.

Sub-step 2.2: Rules for the selection of AICs

These rules aren't executed here because there is no domain model. The rules 6 to 10 apply on patterns with attributes or methods. For an example of execution of these rules, see the "virtual polling system" example.

Sub-step 2.3: Rules for spatio-temporal arrangement of AIOs

For each couple of sister tasks executed into AIOs, we generate an abstractAdjacency relationship between these AIOs. As AIOs can be of two types (i.e., ACs or AICs), there are four possible rules to be applied (Rule 11-14).

Sub-step 2.4: Rules for the definition of abstract dialog control

By analogy with the previous sub-step, for each couple of sister tasks executed into AIOs, we generate an abstractDialogControl relationship between these AIOs that have the same semantics as the temporal relationship defined between the tasks. As AIOs can be of two types (i.e., ACs or AICs), there are four possible combination that are considered by Rules 15-18.

Sub-step 2.5: Rules for the derivation of the AUI to domain mappings

This corresponds to rules 19 and 20. For the same reason as for sub-step 2.2, they aren't executed.

Step 3: From AUI Model to CUI Model

The third step implies a transformational system that is composed of necessary rules for realizing the transition from AUI to CUIs. Only GUI is taken into account (no vocal or multimodal UI), so the modality used to interact with the system is entirely graphical (monomodal UI).

Sub-step 3.1: Reification of AC into CC

Rule 21 creates a GC which will be the **main box** of the UI associated to the AC found one level under the root AC in the abstract hierarchy. This **main box** contains the main window of the UI.

Rule 22 creates a GC of type **box** for each AC contained into an AC that was reified into a **main box**.

Sub-step 3.2: Selection of CICs

The current sub-step generates different GICs depending on the type of facets of the corresponding AICs:

- Generation of an **outputText** and an **inputText** that enable to **insert the name** and the **zipCode**: Rule 23 is applied each time an AIC with an input facet of type create element is encountered.
- Generation of a GC of type **box** that will embed a **group of radio buttons** and a GIC of type **outputText** representing the label associated to this group when an input facet of type select element is encountered: Rule 24; The radio buttons associated to this group are created by applying Rule 25. The rules are used in order to **select the gender** of the user, the **ageCategory** and also his **answers** to the questions

- Generation of a GIC of type **outputText** each time an output facet of type create is encountered. For this purpose Rule 26 has to be applied in order to ensure the display of the **titles of the questions**.
- Generation a button that will ensure the **send questionnaire** task each time when a control facet of type start operation is encountered: rule 27.

Sub-step 3.3: Arrangement of CICs

For each couple of adjacent AIOs that are reified into graphicalCIOs, we define a graphicalAdjacency relationship between these graphicalCIOs. As AIOs can be of two types (i.e., ACs or AICs), there are four possible combination to take into account. For each combination a specific rule is considered: Rules 28-31.

Sub-step 3.4: Navigation definition

The rules that ensure the navigation definition are not applied in the current case study as all the sub-tasks of the virtual polling system are presented combined into the same window.

Sub-step 3.5: Concrete Dialog Control Definition

For each couple of AIOs with a dialog control relationship, a transposition of this relationship to the graphicalCIOs that reify them is realized. As AIOs are of two types (i.e., ACs and AICs), four rules describing the four possible combinations are considered: Rules 32-35.

Sub-step 3.6: Derivation of CUI to Domain Relationship

Rules 36 and 37 are used to transpose the *updates* and *triggers* relationships from the abstract to the concrete level. These relationships map GICs with attributes and methods from the Domain Model.

6.2.1 Currency convertor in ATOM³

The task model created in ATOM³ to represent the currency convertor gives the following graph:

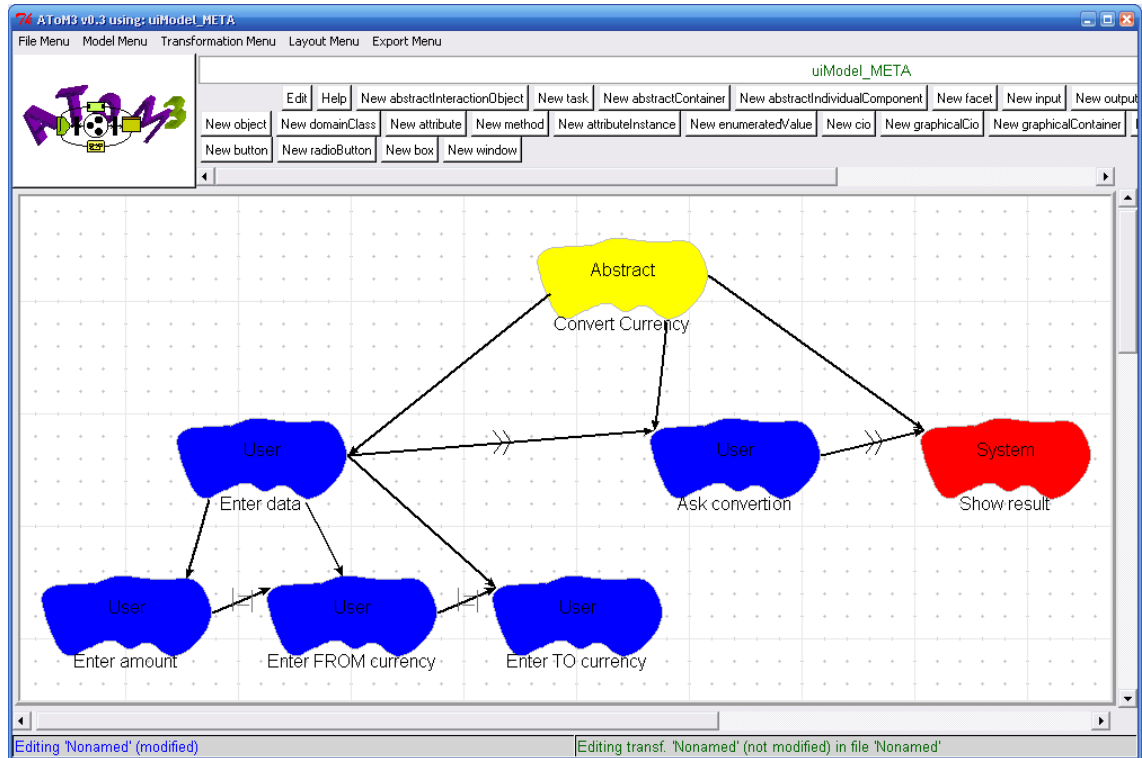


Figure 6-2. Currency convertor task model in ATOM³

The graph in ATOM³ is easy to understand because, here, the relations representations are the same as the UsiXML standard and so, the same as in idealXML (as said before, ATOM³ allows giving any graphical representation to the objects).

We will now apply the rules set described in the point 6.1 (rule set description):

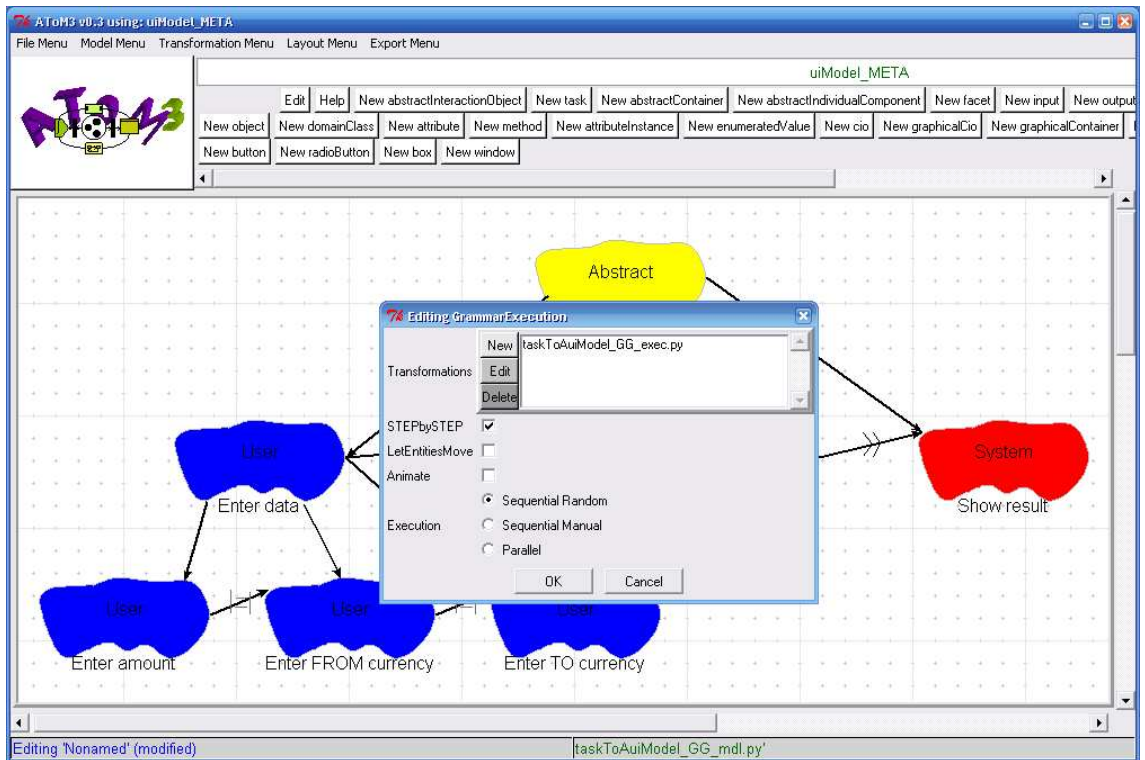


Figure 6-3. Editing grammar execution

As described in chapter 3, we have already generated python code for our transformation rules; ATOM³ thus generated one file for each rules set. Now we have to add this file to the list of rules sets ATOM³ will apply on our model, as we see on figure 6-3. ATOM³ also allows applying the rules step by step to see the effects of each one. So we will show here the screenshots of the model after each rule execution.

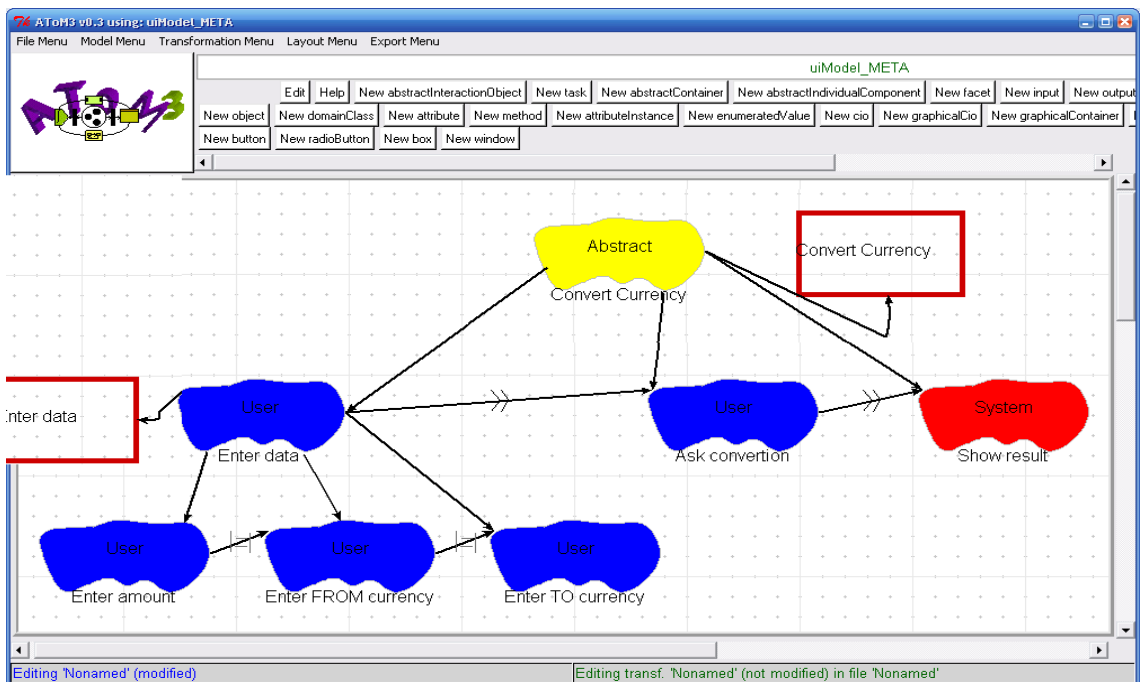


Figure 6-4. Currency convertor after execution of rule 1

After the execution of the first rule (Rule 1: executedINAC, figure 9.1), ATOM³ has created an abstract container for each task that has sub-tasks.

Then, as we see on the figure 6-5, ATOM³ executes the second rule (Rule 2: executedInAIC, figure 9.2) and creates an abstract individual component for each task that has no sub-task.

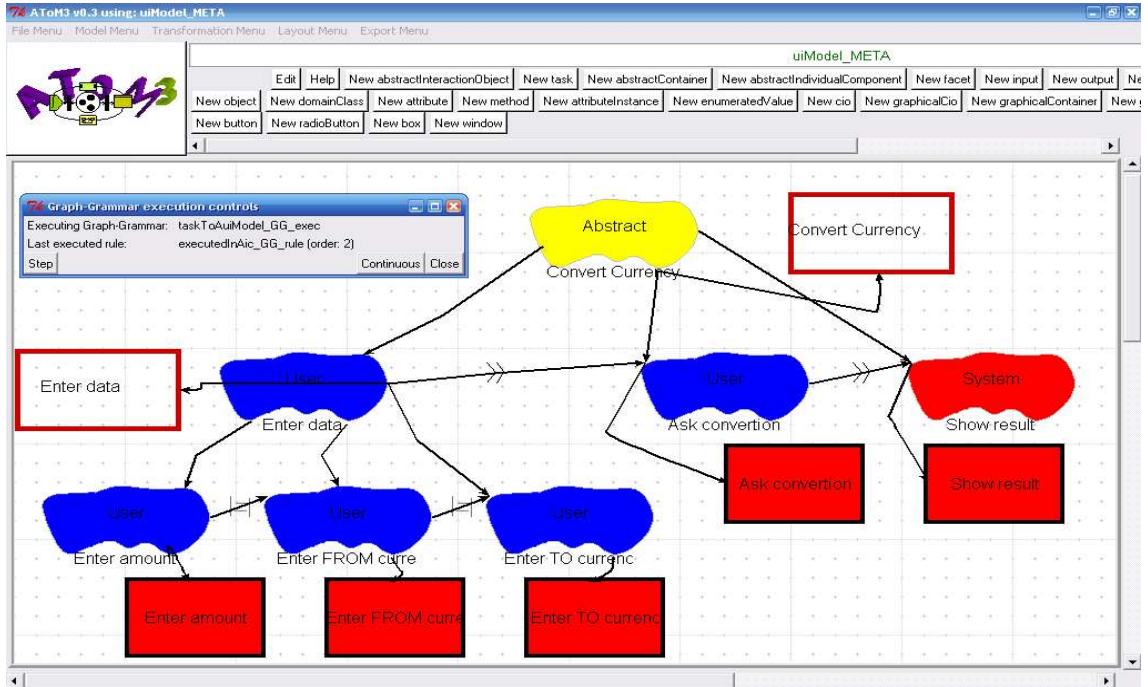


Figure 6-5. Currency convertor after execution of rule 2

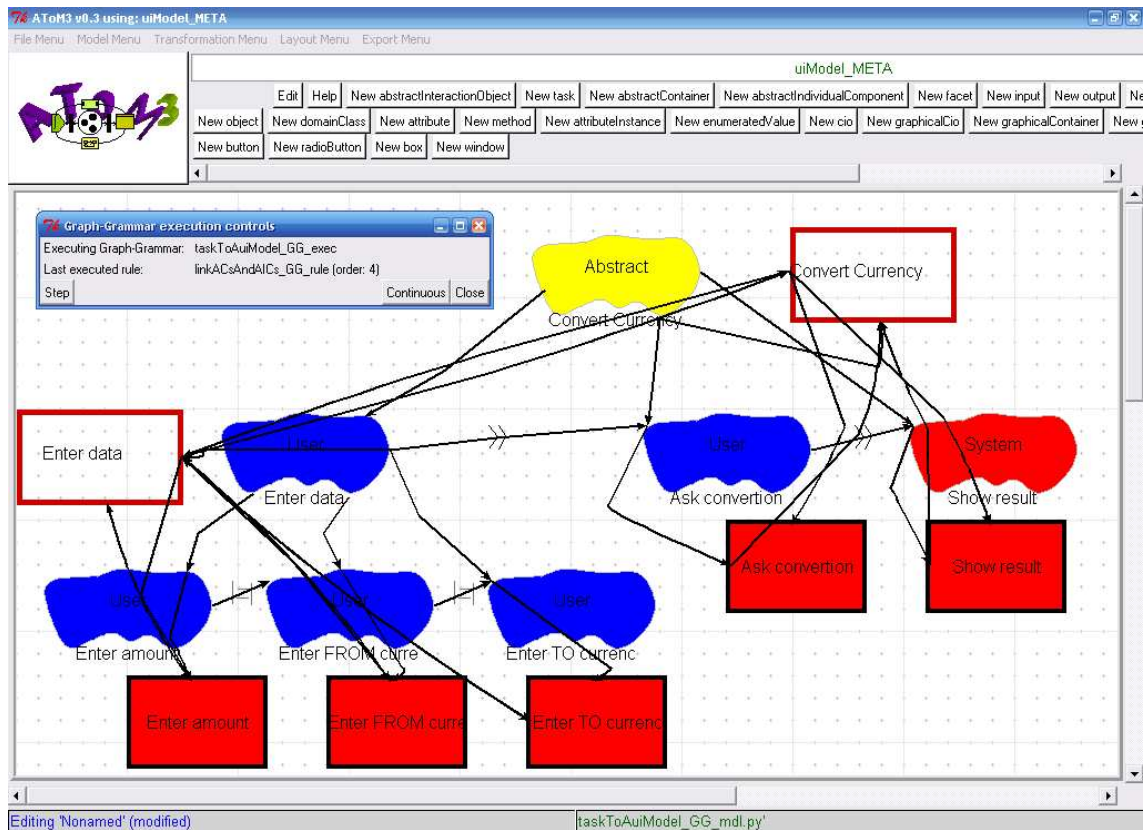


Figure 6-6. Currency convertor after execution of rule 3 and 4

Then, as we see on the figure 6-6, ATOM³ executes the two rules intended to create bidirectional links between abstract objects (abstract containers and abstract individual components) when one contains the other (rules 3 and 4, figure 9-3 and 9-4). We already become to see one of the problems with ATOM³: even small graphs quickly become very hard to read because of the arrows and objects crossing each other, and because the objects do not move to make the graph more readable during the execution of the transformations.

The problem becomes even more obvious on the figure 6-7. Here, for each pair of tasks linked by a dialog control relation, the same relation has been created between the abstract containers or abstract individual component that these tasks are executed in (figure 9-15 to 9-18). It really becomes hard to understand the graph, and the only way of knowing what a relation represents is now clicking on it to see its properties.

On the figure 6-8, abstract adjacency relations have been created between abstract containers of individual components corresponds to tasks linked by an « enabling » relation (rules 14 to 17, figure 9.14 to 9.17).

For this simple example, we didn't implement the domain model, so there is no attribute or methods, and the transformation engine won't thus create any facets for the abstract individual components. This is done in the next example (virtual polling system). We didn't set the user actions of the tasks, so ATOM³ won't create facets for the abstract individual

components either. One the figure 6-9, we see that a box named « main box » has been created and reify the abstract container of the root task, « convert currency ». A window has also been created and is contained by the mainbox (rule 20, figure 9-20).

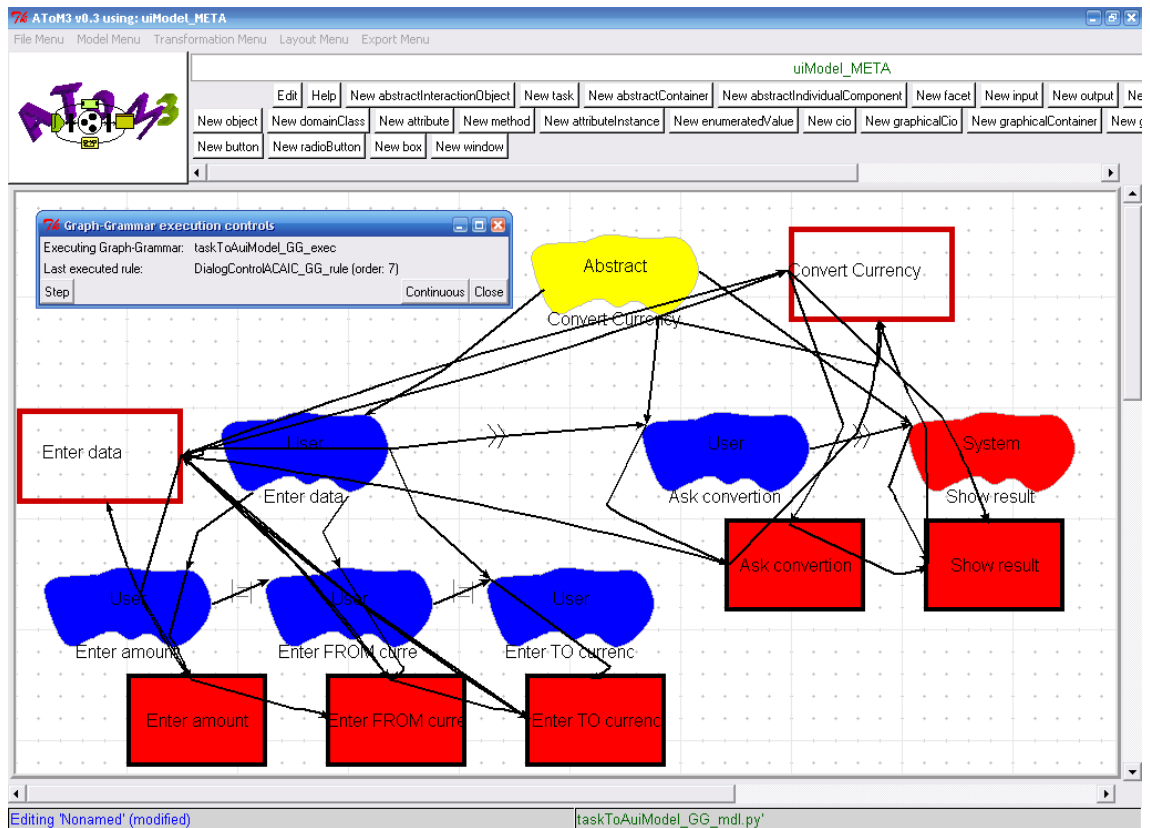


Figure 6-7. Currency convertor after execution of rule 6 and 7

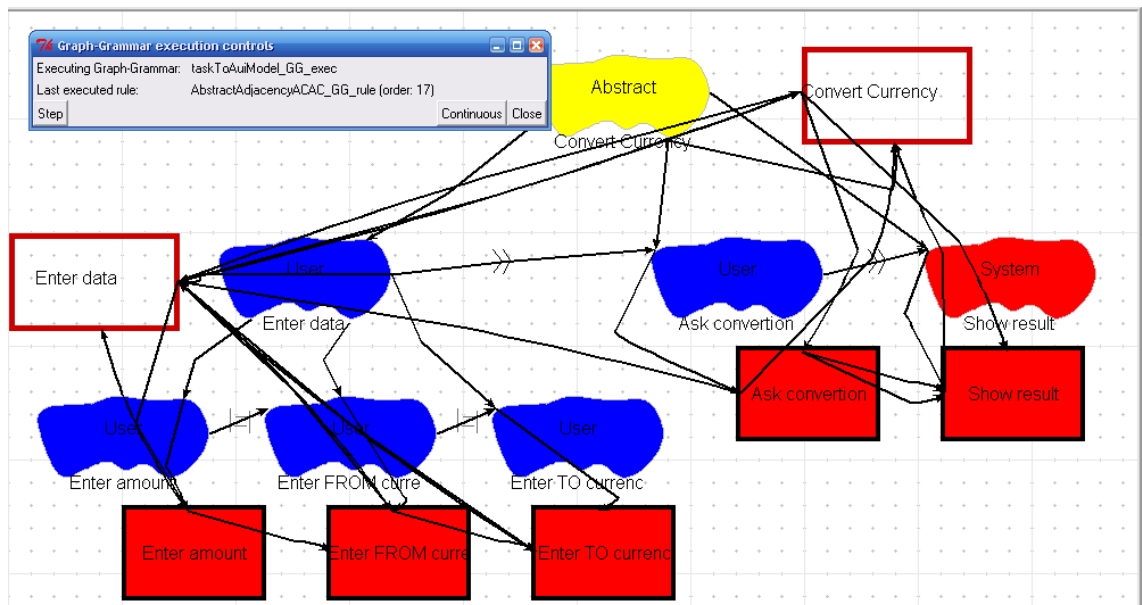


Figure 6-8. Currency convertor after execution of rule 11, 12, 13 and 14 (14, 15, 16 and 17 in ATOM³)

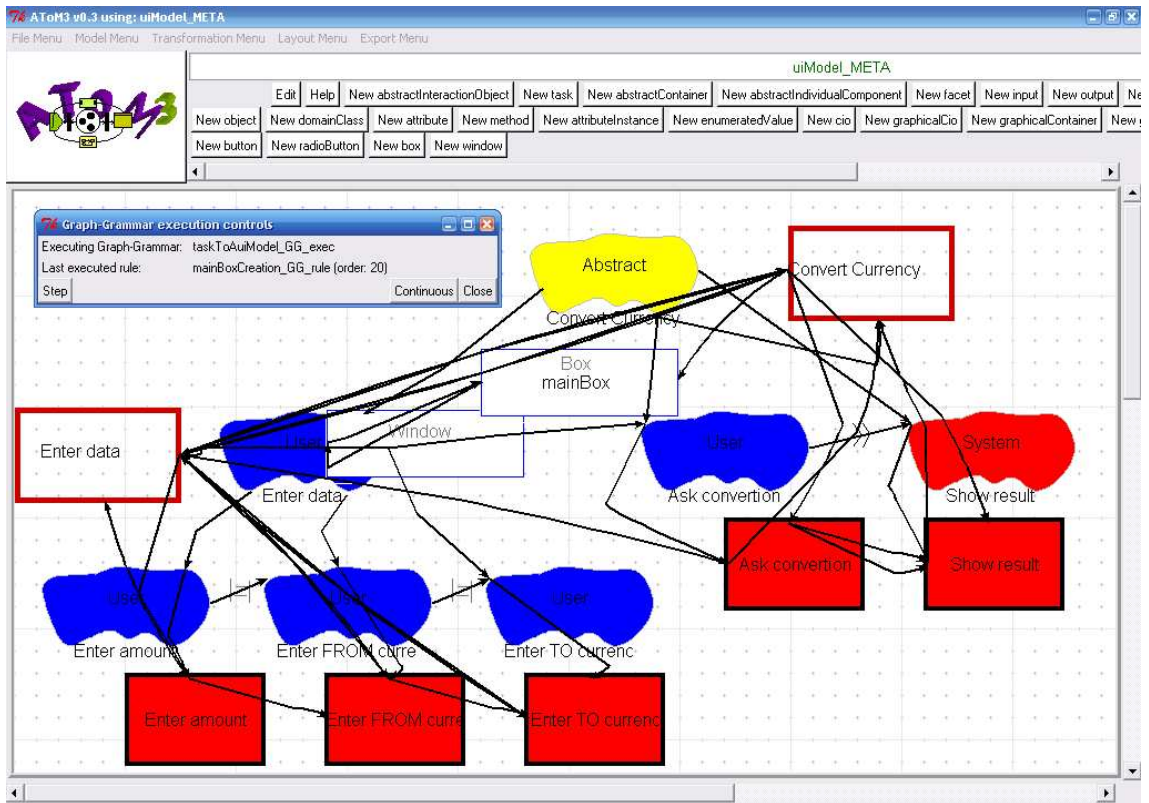


Figure 6-9. Currency convertor after execution of rule 21 (20 in AToM³)

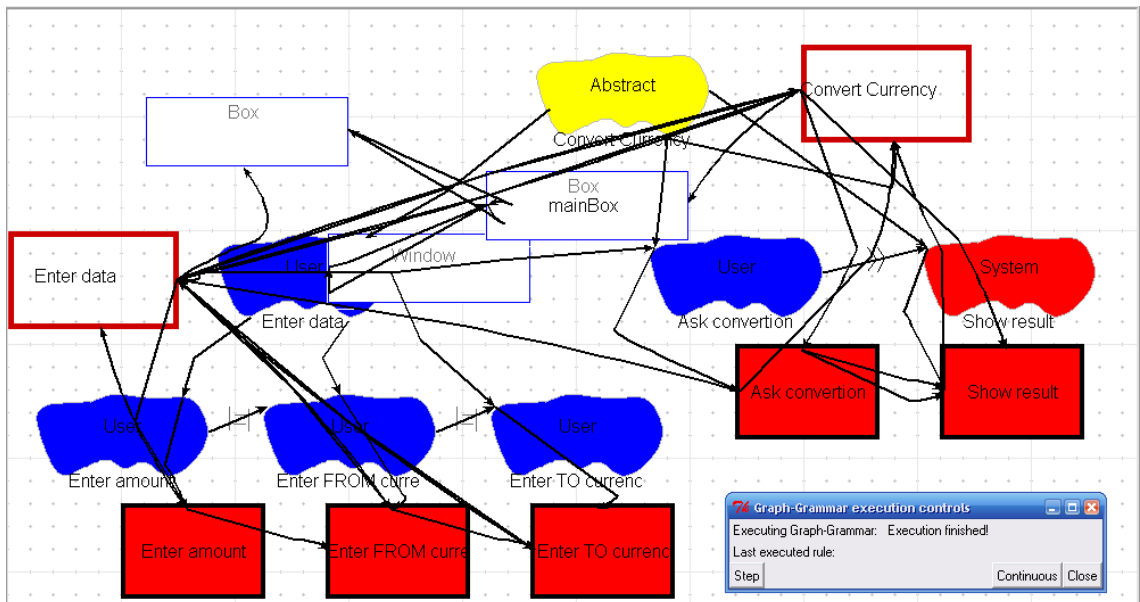


Figure 6-10. Currency convertor after execution of rule 22 (21 in AToM³)

Finally, a box is created for each abstract container contained by another abstract container (figure 6-10).

6.2.2 Custom java transformation engine

Our custom java transformation engine takes directly the UsiXML files as input. This file is described above in the example description.

We launch the application and open the file as « source file ». On the next screen, we see the source file has been opened and is showed in the first text area.

At the right of the figure 6-11, we see the tree of all transformation rules implemented in the program. First, we only select the rules to create the abstract objects, namely « Abstract Interaction Level » in the application.

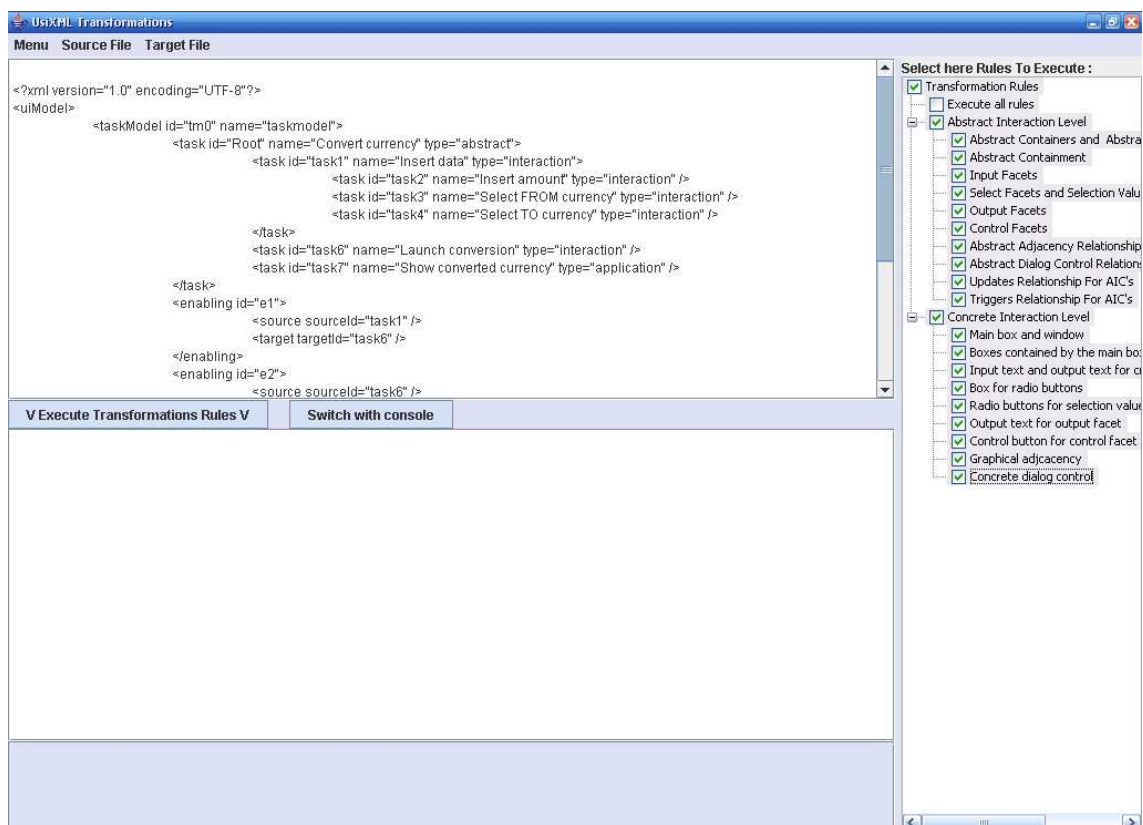
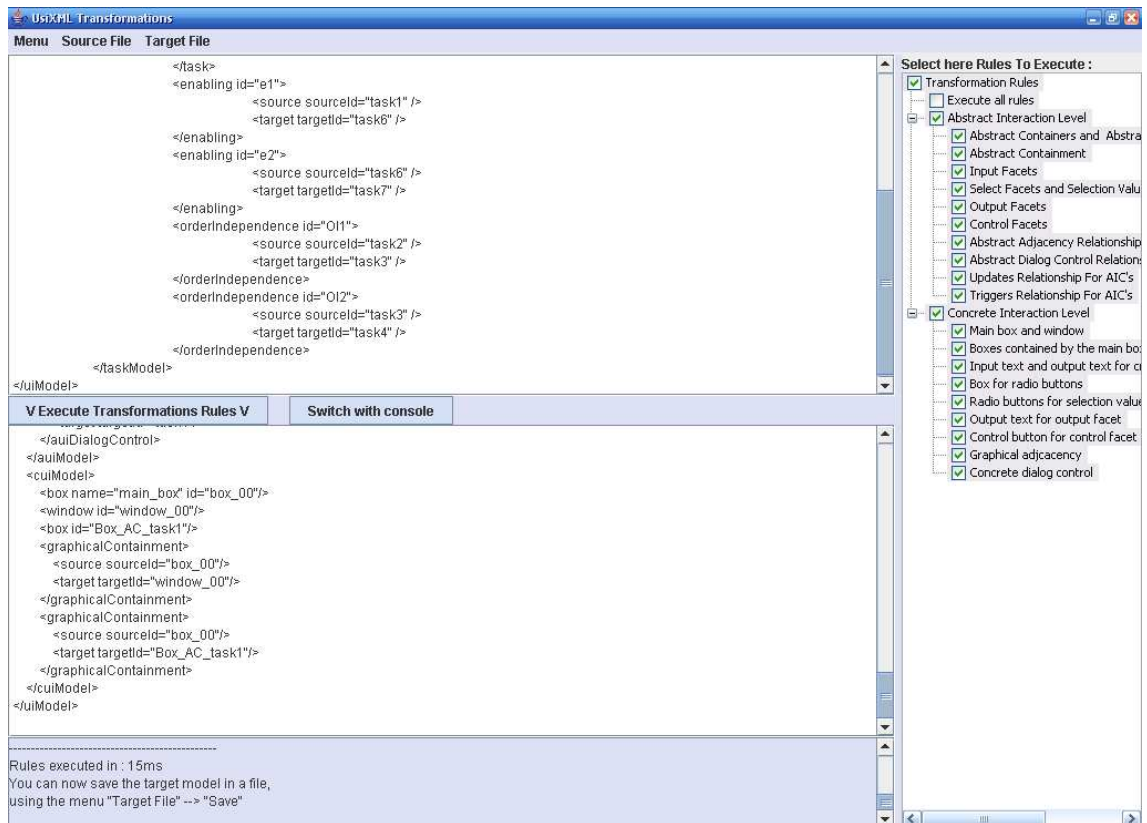


Figure 6-11. Source model and transformation rules selection in custom java transformation engine

We then click on the button « Execute Transformations Rules » located between the two text areas, and the transformation engines applies all the selected rules (of course, the one that can be applied) on the source model, then put the result in the bottom text area.

This result is shown in figure 6-12. We see in the console that the transformation rules have been executed properly, in 15ms time.



6-12. Custom java transformation engine after rules execution

The file resulting from the execution is shown in figure 6-13:

```

<?xml version="1.0" encoding="UTF-8" ?>
<uiModel>
  <taskModel id="tm0" name="taskmodel">
    <task id="Root" name="Convert currency" type="abstract">
      <task id="task1" name="Insert data" type="interaction">
        <task id="task2" name="Insert amount"
          type="interaction"/>
        <task id="task3" name="Select FROM currency"
          type="interaction"/>
        <task id="task4" name="Select TO currency"
          type="interaction"/>
      </task>
      <task id="task6" name="Launch conversion"
        type="interaction"/>
      <task id="task7" name="Show converted currency"
        type="application"/>
    </task>
    <enabling>
      <source sourceId="AC_task1"/>
      <target targetId="AIC_task6"/>
    </enabling>
    <enabling>
      <source sourceId="AIC_task6"/>
      <target targetId="AIC_task7"/>
    </enabling>
    <orderIndependence id="OI1">
      <source sourceId="AIC_task2"/>
      <target targetId="AIC_task3"/>
    </orderIndependence>
  </taskModel>
</uiModel>

```

```

        </orderIndependence>
        <orderIndependence id="OI2">
            <source sourceId="AIC_task3"/>
            <target targetId="AIC_task4"/>
        </orderIndependence>
    </taskModel>
</domainModel/>
<mappingModel>
    <isReifiedBy>
        <source sourceId="AC_Root"/>
        <target targetId="box_00"/>
    </isReifiedBy>
    <isReifiedBy>
        <source sourceId="AC_task1"/>
        <target targetId="Box_AC_task1"/>
    </isReifiedBy>
    <isReifiedBy>
        <source sourceId="AIC_task6"/>
        <target targetId="Box_AIC_task6"/>
    </isReifiedBy>
    <isReifiedBy>
        <source sourceId="AIC_task7"/>
        <target targetId="Box_AIC_task7"/>
    </isReifiedBy>
</mappingModel>
<auiModel>
    <abstractContainer name="Convert currency" id="AC_Root">
        <abstractContainer name="Insert data" id="AC_task1">
            <abstractIndividualComponent name="Insert amount"
                id="AIC_task2"/>
            <abstractIndividualComponent name="Select FROM
                currency" id="AIC_task3"/>
            <abstractIndividualComponent name="Select TO currency"
                id="AIC_task4"/>
        </abstractContainer>
        <abstractIndividualComponent name="Launch conversion"
            id="AIC_task6"/>
        <abstractIndividualComponent name="Show converted
            currency" id="AIC_task7"/>
    </abstractContainer>
    <abstractContainment id="ACont_AC_task1">
        <source sourceId="AC_task1"/>
        <target targetId="AIC_task2"/>
        <target targetId="AIC_task3"/>
        <target targetId="AIC_task4"/>
    </abstractContainment>
    <abstractContainment id="ACont_AC_Root">
        <source sourceId="AC_Root"/>
        <target targetId="AC_task1"/>
        <target targetId="AIC_task6"/>
        <target targetId="AIC_task7"/>
    </abstractContainment>
    <abstractAdjacency id="AA_AC_task1">
        <source sourceId="AC_task1"/>
        <target targetId="AIC_task6"/>
    </abstractAdjacency>
    <abstractAdjacency id="AA_AIC_task6">
        <source sourceId="AIC_task6"/>
        <target targetId="AIC_task7"/>
    </abstractAdjacency>

```

```

<audiDialogControl control="EnablingType" id="AuiDCnull">
  <source sourceId="AC_task1"/>
  <target targetId="AIC_task6"/>
</audiDialogControl>
<audiDialogControl control="EnablingType" id="AuiDCnull">
  <source sourceId="AIC_task6"/>
  <target targetId="AIC_task7"/>
</audiDialogControl>
<audiDialogControl control="OrderIndependenceType"
  id="AuiDCOI1">
  <source sourceId="AIC_task2"/>
  <target targetId="AIC_task3"/>
</audiDialogControl>
<audiDialogControl control="OrderIndependenceType"
  id="AuiDCOI2">
  <source sourceId="AIC_task3"/>
  <target targetId="AIC_task4"/>
</audiDialogControl>
</audiModel>
<cuiModel>
  <box name="main_box" id="box_00"/>
  <window id="window_00"/>
  <box id="Box_AC_task1"/>
  <box id="Box_AIC_task6"/>
  <box id="Box_AIC_task7"/>
  <graphicalContainment>
    <source sourceId="box_00"/>
    <target targetId="window_00"/>
  </graphicalContainment>
  <graphicalContainment>
    <source sourceId="box_00"/>
    <target targetId="Box_AC_task1"/>
  </graphicalContainment>
  <graphicalContainment>
    <source sourceId="box_00"/>
    <target targetId="Box_AIC_task6"/>
  </graphicalContainment>
  <graphicalContainment>
    <source sourceId="box_00"/>
    <target targetId="Box_AIC_task7"/>
  </graphicalContainment>
  <graphicalAdjacency>
    <source sourceId="Box_AC_task1"/>
    <target targetId="Box_AIC_task6"/>
  </graphicalAdjacency>
  <graphicalAdjacency>
    <source sourceId="Box_AIC_task6"/>
    <target targetId="Box_AIC_task7"/>
  </graphicalAdjacency>
  <cuiDialogControl>
    <symbol>EnablingType</symbol>
    <source sourceId="Box_AC_task1"/>
    <target targetId="Box_AIC_task6"/>
  </cuiDialogControl>
  <cuiDialogControl>
    <symbol>EnablingType</symbol>
    <source sourceId="Box_AIC_task6"/>
    <target targetId="Box_AIC_task7"/>
  </cuiDialogControl>
</cuiModel>

```



```
</uiModel>
```

Figure 6-14 : execution result on the currency convertor

As we can see, the UsiXML syntax is respected, and the file is automatically indented. The transformation engine created a mapping model, an abstract user interface model, and a concrete user interface model. Because there is no domain model in our example, no facet has been created for the abstract individual objects, as said before for AToM³.

6.3 Use case: Polling System

The virtual polling system based is based on a web interface. Here is the complete description of the example and the transformation rules needed [Limbourg, Stanciulescu]

This example is more complete than the previous one because here, task, domain and mapping models have been implemented, which leads to the execution of more transformation rules.

Here is the detailed description of the polling system [Limbourg, Stanciulescu]:

This case study applies our transformational approach for developing a UI on an opinion polling system aiming at collecting opinions of users regarding a certain subject. The scenario of this case study (Figure 5-1) is the following: from the Task and Domain Models, an AUI is produced, from which GUIs is derived.

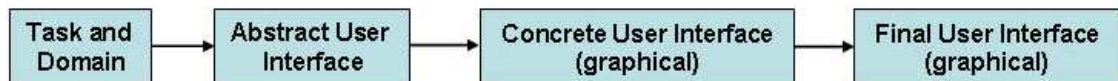


Figure 6-15. Polling system transformation approach

Step 1: The Task and Domain Models

The task model, the domain model and the mappings between them are graphically described using IdealXML tool. The upper part of Figure 5-2 depicts a CTT representation of the task model envisioned for the future system. The root task consists of participating to an opinion poll. The user has to provide her personal data like name, zip code, gender, age category. Further, the user iteratively answers some questions. Answering a question is composed of a system task showing the title of the question and of an interactive task consisting in selecting one answer among several proposed ones. Once the questions are answered, the questionnaire is sent back to its initiator. The bottom part of Figure 5-2 illustrates the domain model of our UI. The domain model has the appearance of a class diagram and can be described as follow: a participant participates to a questionnaire, a questionnaire is made of several questions and a question is attached to a series of answers.

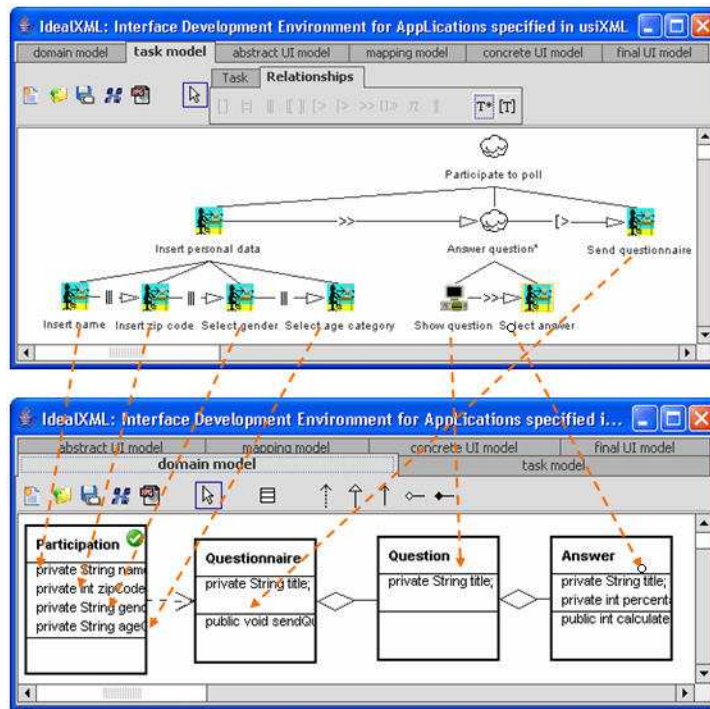


Figure 6-16. Mappings between the Task Model and the Domain Model

The dashed arrows between the two models in Fig. 5-2 depict the mappings relationships between the elements of the *Task* and the *Domain Model*. The sub-tasks of **Insert personal data** task is mapped onto the correspondent attributes of **Participation** class (**name**, **zipCode**, **gender** and **ageCategory**). **Show question** is mapped onto the attribute title of class **Question**. The task **Select answer** is mapped onto the attribute title of the class **Answer**. Finally, the task **Send questionnaire** is mapped onto the method **sendQuestionnaire** of the class **Questionnaire**. Figure 5-3 illustrates the graphical editor of the *Mapping Model* in IdealXML tool. Each leaf tasks is mapped on the corresponding attribute or method of the classes contained in the *Domain Model*.

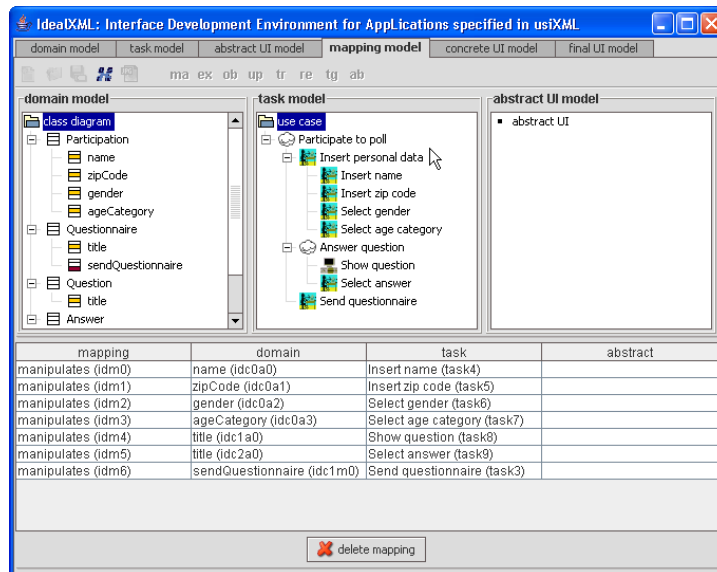


Figure 6-17. Mapping model for the virtual polling system

IdealXML generates automatically the UsiXML specifications for the *Task*, *Domain* and *Mapping Models*. Figure 5-4 describes the UsiXML specification corresponding to the *Task Model*. The first 14 lines describe the hierarchical decomposition of the *Task Model*, while lines 15 to 43 describe the relationships between the tasks.

```

1<taskModel id="TaskModelCS1" name="TaskModel">
2  <task id="Root" name="Participate to poll" importance="5" type="abstract">
3    <task id="T1" name="Insert personal data" importance="3" type="interactive">
4      <task id="T11" name="Insert name" userAction="create" taskItem="element" importance="5" type="interactive"/>
5      <task id="T12" name="Insert zip code" userAction="create" taskItem="element" importance="5" type="interactive"/>
6      <task id="T13" name="Select gender" userAction="select" taskItem="element" importance="5" type="interactive"/>
7      <task id="T14" name="Select age category" userAction="select" taskItem="element" importance="5" type="interactive"/>
8    </task>
9    <task id="T2" name="Answer question" importance="3" type="abstract">
10     <task id="T21" name="Show question" userAction="create" taskItem="collection" importance="5" type="system"/>
11     <task id="T22" name="Select answer" userAction="select" taskItem="element" importance="5" type="interactive"/>
12    </task>
13    <task id="T3" name="Send questionnaire" userAction="start" taskItem="operation" importance="3" type="interactive"/>
14  </task>
15  <enabling id="e1">
16    <source sourceId="T1">
17      <target targetId="T2">
18    </enabling>
19  </enabling id="e2">
20    <source sourceId="T2">
21      <target targetId="T3">
22    </disabling>
23  </independentConcurrency id="e11">
24    <source sourceId="T11">
25      <target targetId="T12">
26  </independentConcurrency>
27  </independentConcurrency id="e12">
28    <source sourceId="T12">
29      <target targetId="T13">
30  </independentConcurrency>
31  </independentConcurrency id="e13">
32    <source sourceId="T13">
33      <target targetId="T14">
34  </independentConcurrency>
35  </iteration id="e3">
36    <source sourceId="T2">
37      <target targetId="T2">
38  </iteration>
39  </enabling id="e21">
40    <source sourceId="T21">
41      <target targetId="T22">
42  </enabling>
43</taskModel>

```

Figure 6-18. Task model expressed in UsiXML

Figure 5-5 illustrates the *Domain Model* expressed in UsiXML. Lines 1 to 31 define the classes that are involved into the class diagram. Lines 9 to 12 describe the attribute “ageCategory” that can have different values expressed under the form of an enumerated domain (the possible values are “18-35”, “35-45”, “more than 45”). Lines 27 to 30 define a method with its two parameters i.e., an input and an output parameter). Lines 32 to 44 describe the relationships between the above described classes.

```

1 <domainModel id="domainModelCS2" name="domainModel">
2   <domainClass id="DC1" name="Participation">
3     <attribute id="A1DC1" name="name" attributeDataType="string" attributeCardMin="1" attributeCardMax="1"/>
4     <attribute id="A2DC1" name="zipCode" attributeDataType="integer" attributeCardMin="1" attributeCardMax="1"/>
5     <attribute id="A3DC1" name="gender" attributeDataType="string" attributeCardMin="1" attributeCardMax="1"/>
6       <enumeratedValue name="Male"/>
7       <enumeratedValue name="Female"/>
8     </attribute>
9     <attribute id="A4DC1" name="ageCategory" attributeDataType="string" attributeCardMin="1" attributeCardMax="1"/>
10       <enumeratedValue name="18-35"/>
11       <enumeratedValue name="35-45"/>
12       <enumeratedValue name="45+"/>
13     </attribute>
14   </domainClass>
15   <domainClass id="DC2" name="Questionnaire">
16     <attribute id="A1DC2" name="title" attributeDataType="string" attributeCardMin="1" attributeCardMax="1"/>
17     <method id="M1DC2" name="sendQuestionnaire">
18       <param id="P1M1DC1" dataType="Questionnaire" name="qu" paramType="input"/>
19     </method>
20   </domainClass>
21   <domainClass id="DC3" name="Question">
22     <attribute id="A1DC3" name="title" attributeDataType="string" attributeCardMin="1" attributeCardMax="1"/>
23   </domainClass>
24   <domainClass id="DC4" name="Answer">
25     <attribute id="A1DC4" name="title" attributeDataType="string" attributeCardMin="1" attributeCardMax="1"/>
26       <enumeratedValue name="Question1" />
27       <enumeratedValue name="Question2" />
28     </attribute>
29     <attribute id="A2DC4" name="percentage" attributeDataType="integer" attributeCardMin="0" attributeCardMax="1"/>
30     <method id="M1DC4" name="calculatePercentage">
31       <param id="P1M1DC4" dataType="Question" name="qu" paramType="input"/>
32       <param id="P2M1DC4" dataType="integer" name="qu" paramType="output"/>
33     </method>
34   </domainClass>
35   <adHoc id="DA1" name="participation" roleACardMin="0" roleACardMax="n" roleBCardMin="0" roleBCardMax="n">
36     <source sourceId="DC1"/>
37     <target targetId="DC2"/>
38   </adHoc>
39   <aggregation id="DA2" roleACardMin="1" roleACardMax="n" roleBCardMin="1" roleBCardMax="1">
40     <source sourceId="DC2"/>
41     <target targetId="DC3"/>
42   </aggregation>
43   <aggregation id="DA3" roleACardMin="1" roleACardMax="n" roleBCardMin="1" roleBCardMax="1">
44     <source sourceId="DC3"/>
45     <target targetId="DC4"/>
46   </aggregation>
47 </domainModel>

```

Figure 6-19. Domain Model expressed in UsiXML

Figure 5-6 illustrates the mappings established between the *Task Model* and the *Domain Model*. These mappings are specified in UsiXML with the use of two tags (i.e., <source> and <target>) that identify which task will manipulate which attribute/method from the domain model.

```

1 <mappingModel id="MappingDomainCS2" name="mappingDomain">
2   <manipulates id="MA1">
3     <source sourceId="T11">
4       <target targetId="A1DC1"/>
5     </manipulates>
6   <manipulates id="MA2">
7     <source sourceId="T12">
8       <target targetId="A2DC1"/>
9     </manipulates>
10  <manipulates id="MA3">
11    <source sourceId="T13">
12      <target targetId="A3DC1"/>
13    </manipulates>
14  <manipulates id="MA4">
15    <source sourceId="T14">
16      <target targetId="A4DC1"/>
17    </manipulates>
18  <manipulates id="MA5">
19    <source sourceId="T21">
20      <target targetId="A1DC3"/>
21    </manipulates>
22  <manipulates id="MA6">
23    <source sourceId="T22">
24      <target targetId="A1DC4"/>
25    </manipulates>
26  <manipulates id="MA7">
27    <source sourceId="T3">
28      <target targetId="M1DC2"/>
29    </manipulates>
30 </mappingModel>

```

Figure 6-20. Mapping Model expressed in UsiXML

Step 2: From Task and Domain Models to AUI Model

The second transformation step involves a transformation system that contains rules applied in order to realize the transition from the task and domain model to the abstract model. This step is subdivided into five sub-steps according to [Limb04b]. We are improving this work by offering the complete set of rules and by adapting them to the needs of a multimodal UI.

Sub-step 2.1: Rules for the identification of AUI structure

Rules 1 and 2 create abstract containers for tasks that have task children and abstract individual components for leaf tasks. Tasks 4 and 5 reconstruct the containment relationships for these AC's and AIC's.

The result of the application of these rules over the task model structure consists in a hierarchical decomposition of the AUI into abstract containers and abstract individual components.

Sub-step 2.2: Rules for the selection of AICs

The current sub-step generates facets for AICs that support the execution of the leaf task:

- Input facet of type create element for **create name** and **create zipCode** tasks: Rule 6
- Input facet of type select element for **select gender**, **select ageCategory** and **select Answer** tasks: Rule 7; for each enumerated value of an attribute, a selection value with

the same name as the enumerated value, will be attached to the above created facet:
Rule 8

- Output facet of type convey element for the AIC assigned to the task **Show Question Title**: Rule 9
- Control facet of type start operation for the **Send Questionnaire** task: rule 10.

Sub-step 2.3: Rules for spatio-temporal arrangement of AIOs

For each couple of sister tasks executed into AIOs, we generate an abstractAdjacency relationship between these AIOs. As AIOs can be of two types (i.e., ACs or AICs), there are four possible rules to be applied (Rule 11-14).

Sub-step 2.4: Rules for the definition of abstract dialog control

By analogy with the previous sub-step, for each couple of sister tasks executed into AIOs, we generate an abstractDialogControl relationship between these AIOs that have the same semantics as the temporal relationship defined between the tasks. As AIOs can be of two types (i.e., ACs or AICs), there are four possible combination that are considered by Rules 15-18.

Sub-step 2.5: Rules for the derivation of the AUI to domain mappings

In order to ensure the synchronization between the AICs and attributes of objects from the Domain Model, Rule 19 generates the **updates** relationship. Moreover, Rule 20 enables the triggering of methods by AICs through the **triggers** relationship.

Step 3: From AUI Model to CUI Model

The third step implies a transformational system that is composed of necessary rules for realizing the transition from AUI to CUIs. Only GUI is taken into account (no vocal or multimodal UI), so the modality used to interact with the system is entirely graphical (monomodal UI).

For the generation of GUIs the designer takes into consideration just the abstract and concrete graphical part of each transformation rule.

Sub-step 3.1: Reification of AC into CC

Rule 21 creates a GC which will be the **main box** of the UI associated to the AC found one level under the root AC in the abstract hierarchy. This **main box** contains the main window of the UI.

Rule 22 creates a GC of type **box** for each AC contained into an AC that was reified into a **main box**.

Sub-step 3.2: Selection of CICs

The current sub-step generates different GICs depending on the type of facets of the corresponding AICs:

- Generation of an **outputText** and an **inputText** that enable to **insert the name** and the **zipCode**: Rule 23 is applied each time an AIC with an input facet of type create element is encountered.

- Generation of a GC of type **box** that will embed a **group of radio buttons** and a GIC of type **outputText** representing the label associated to this group when an input facet of type select element is encountered: Rule 24; The radio buttons associated to this group are created by applying Rule 25. The rules are used in order to **select the gender** of the user, the **ageCategory** and also his **answers** to the questions
- Generation of a GIC of type **outputText** each time an output facet of type create is encountered. For this purpose Rule 26 has to be applied in order to ensure the display of the **titles of the questions**.
- Generation a button that will ensure the **send questionnaire** task each time when a control facet of type start operation is encountered: rule 27.

Sub-step 3.3: Arrangement of CICs

For each couple of adjacent AIOs that are reified into graphicalCICs, we define a graphicalAdjacency relationship between these graphicalCICs. As AIOs can be of two types (i.e., ACs or AICs), there are four possible combination to take into account. For each combination a specific rule is considered: Rules 28-31.

Sub-step 3.4: Navigation definition

The rules that ensure the navigation definition are not applied in the current case study as all the sub-tasks of the virtual polling system are presented combined into the same window.

Sub-step 3.5: Concrete Dialog Control Definition

For each couple of AIOs with a dialog control relationship, a transposition of this relationship to the graphicalCICs that reify them is realized. As AIOs are of two types (i.e., ACs and AICs), four rules describing the four possible combinations are considered: Rules 32-35.

Sub-step 3.6: Derivation of CUI to Domain Relationship

Rules 36 and 37 are used to transpose the *updates* and *triggers* relationships from the abstract to the concrete level. These relationships map GICs with attributes and methods from the Domain Model.

6.3.1 ATOM³

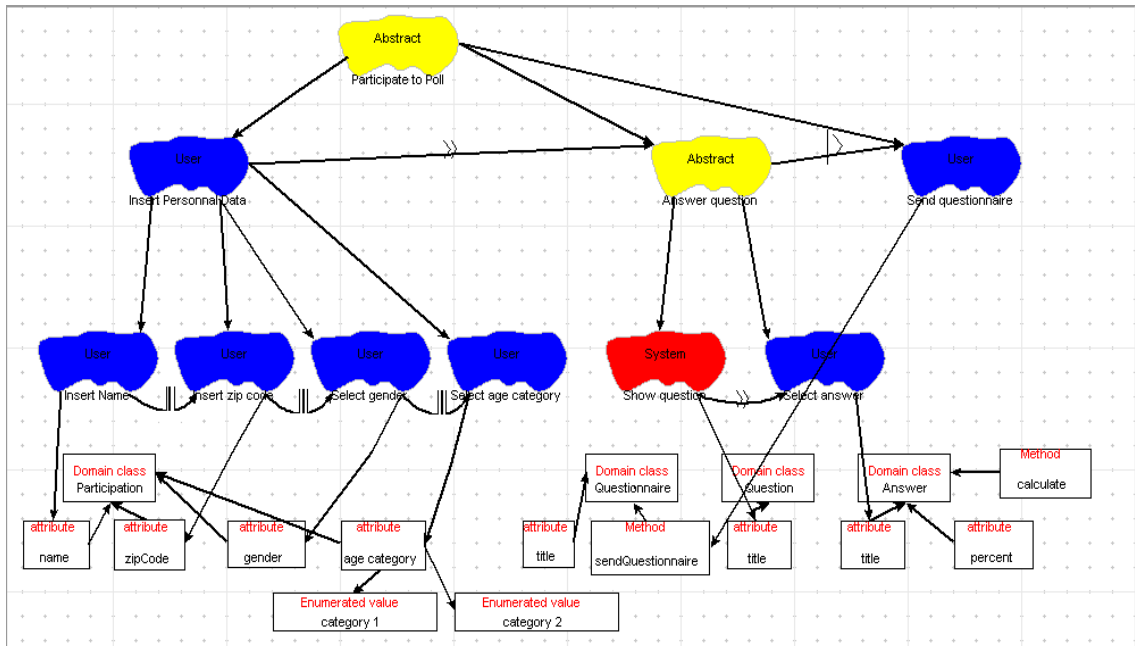


Figure 6-21. Virtual Polling System source model in ATOM³

Figure 6-21 is the ATOM³ representation of the Polling System example.

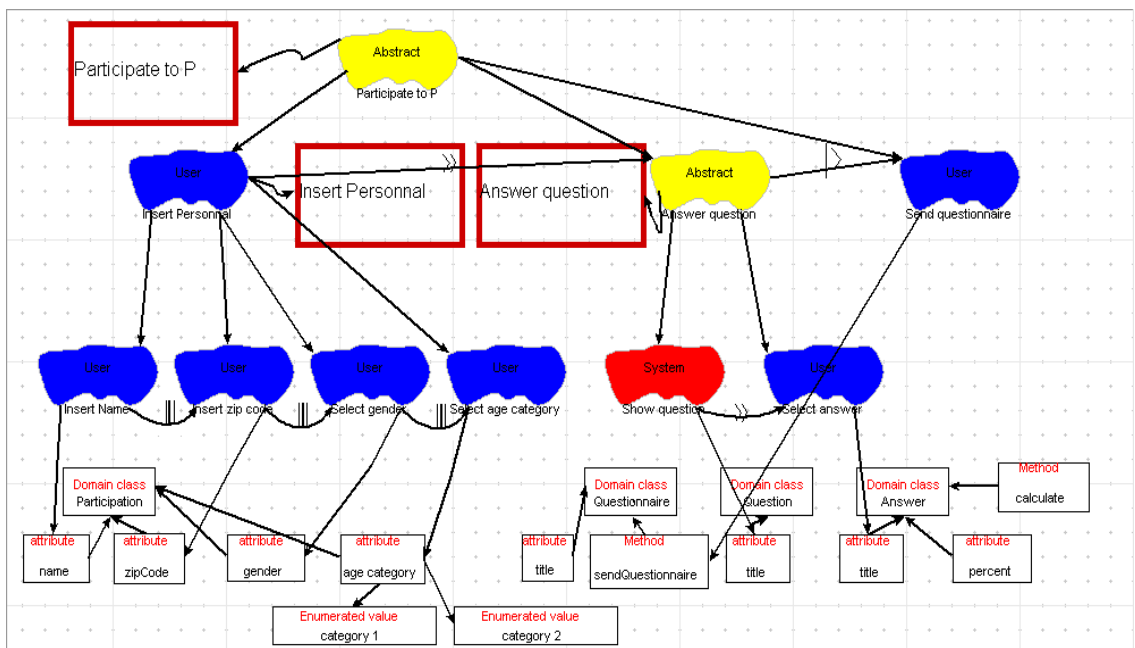


Figure 6-22. Virtual Polling System after execution of rule 1

For each task that has sub-tasks, an *abstract container* has been created with the same name (Figure 6-22).

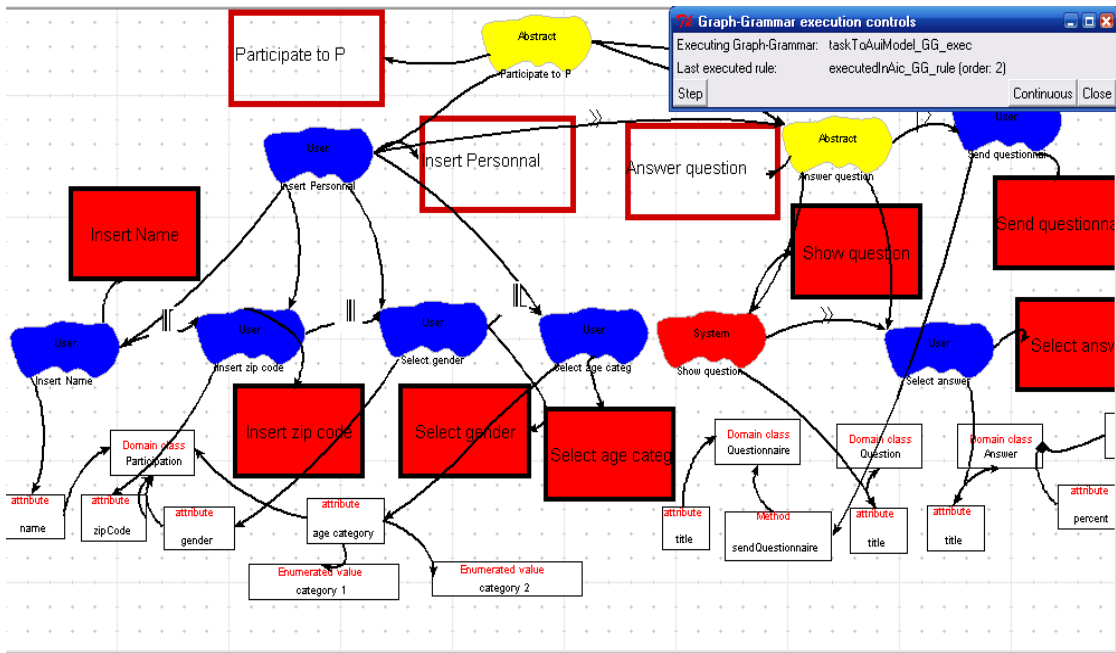


Figure 6-23. Virtual Polling System after execution of rule 2

On figure 6-23 we see that an *abstract individual component* has been created for each task that doesn't have sub-tasks. As we can see, we are only at the beginning of the transformation set, and the graph already becomes unreadable.

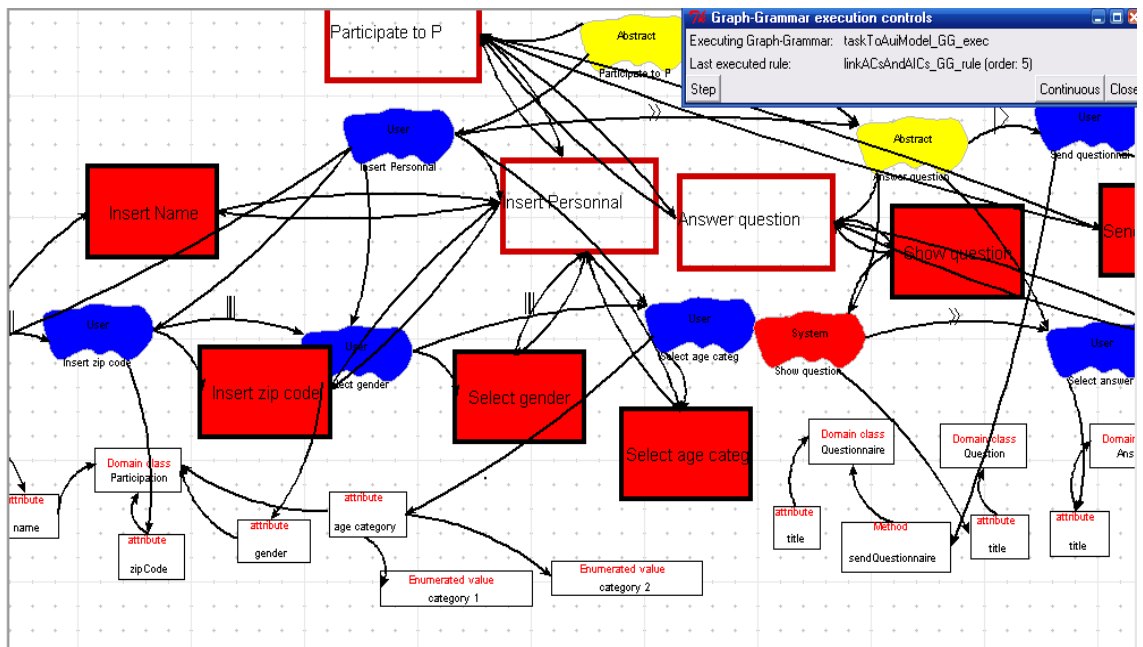


Figure 6-24. Virtual Polling System after execution of rule 4 and 5

Bidirectional links (“contains” and “is contained by”) have been created between AC’s and the AC’s or AIC’s they contain on the figure 6-24.

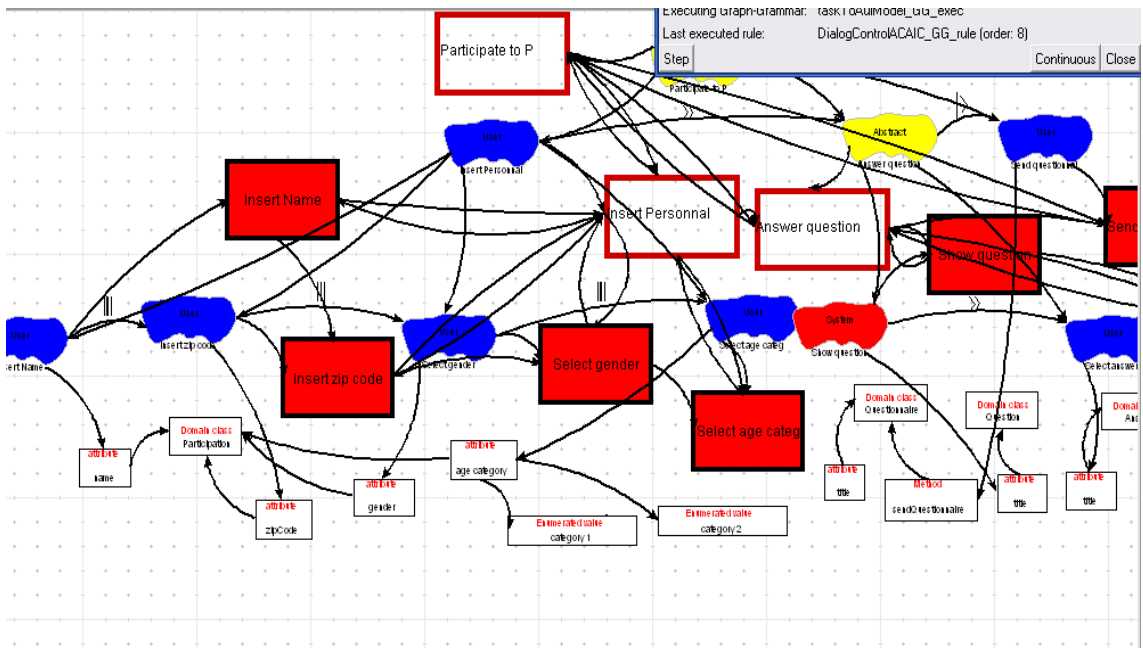


Figure 6-25. Virtual Polling System after execution of rule 15 to 18

Figure 6-25: For each task of type “create” that manipulates an attribute, an input facet of type create has been created.

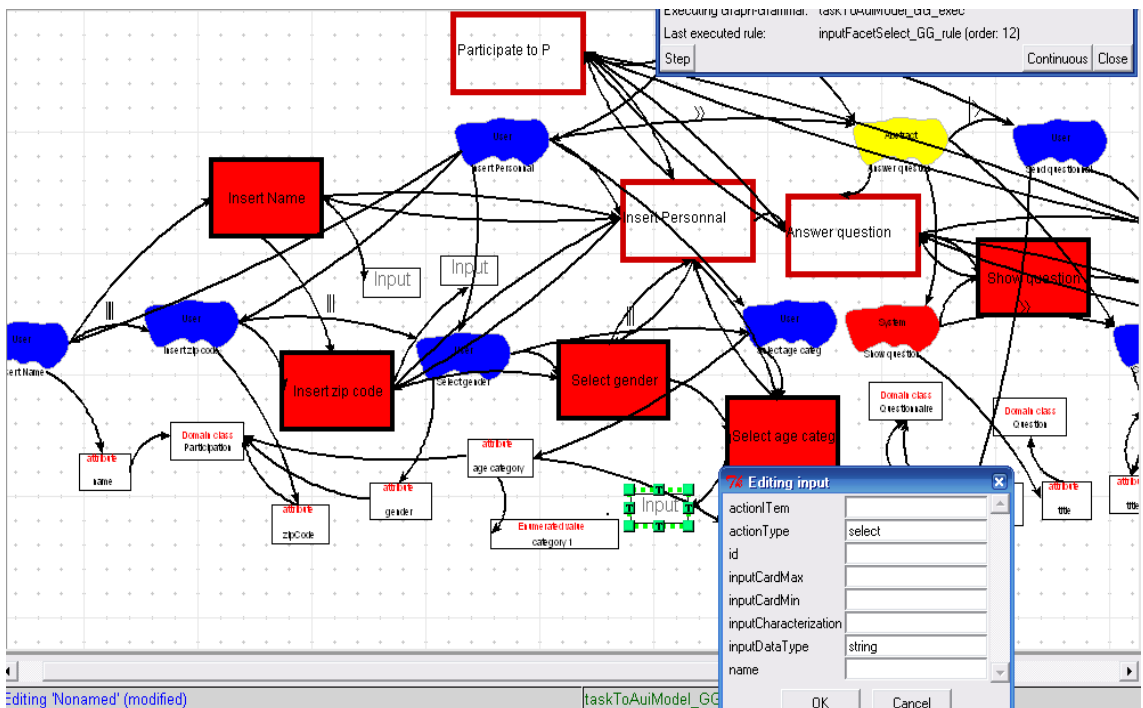


Figure 6-26. Virtual Polling System after execution of rule 6

On figure 6-26 we see that for each task of type “select” that manipulates an attribute, an input facet of type create has been created (for the sake of readability, the properties window of the input has been open here, and we see it is well an input with action type “select”).

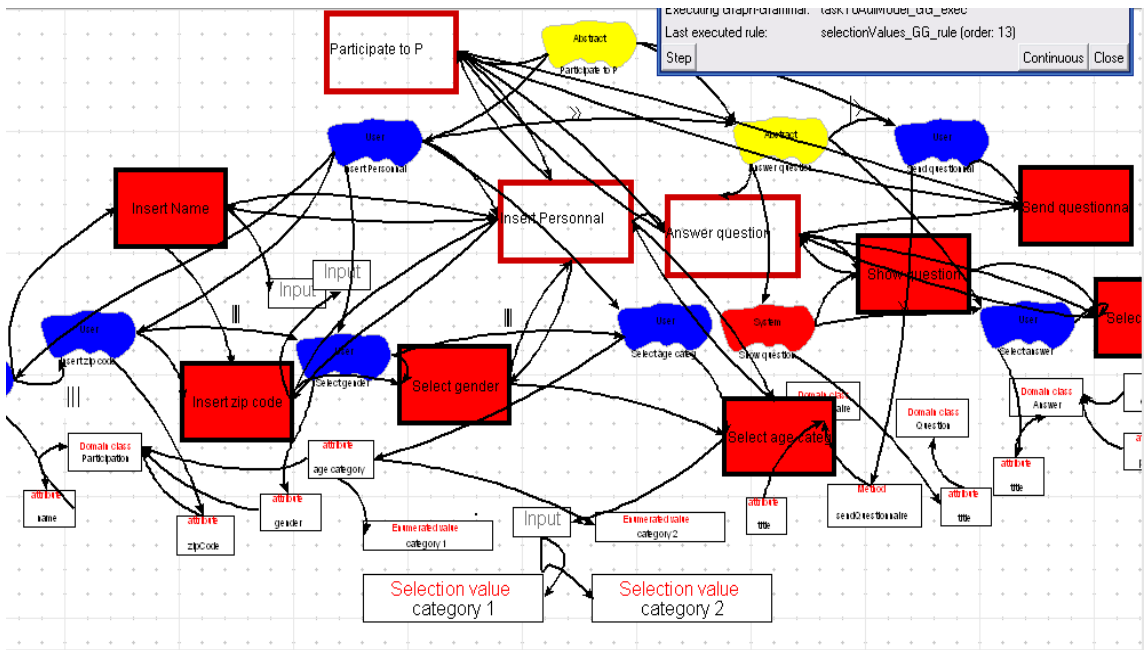


Figure 6-27. Virtual Polling System after execution of rule 8

For each *Enumerated value* linked to an attribute, a *selection value* is created and linked to the AIC corresponding to the (select) task that manipulates the attribute (Figure 6-27).

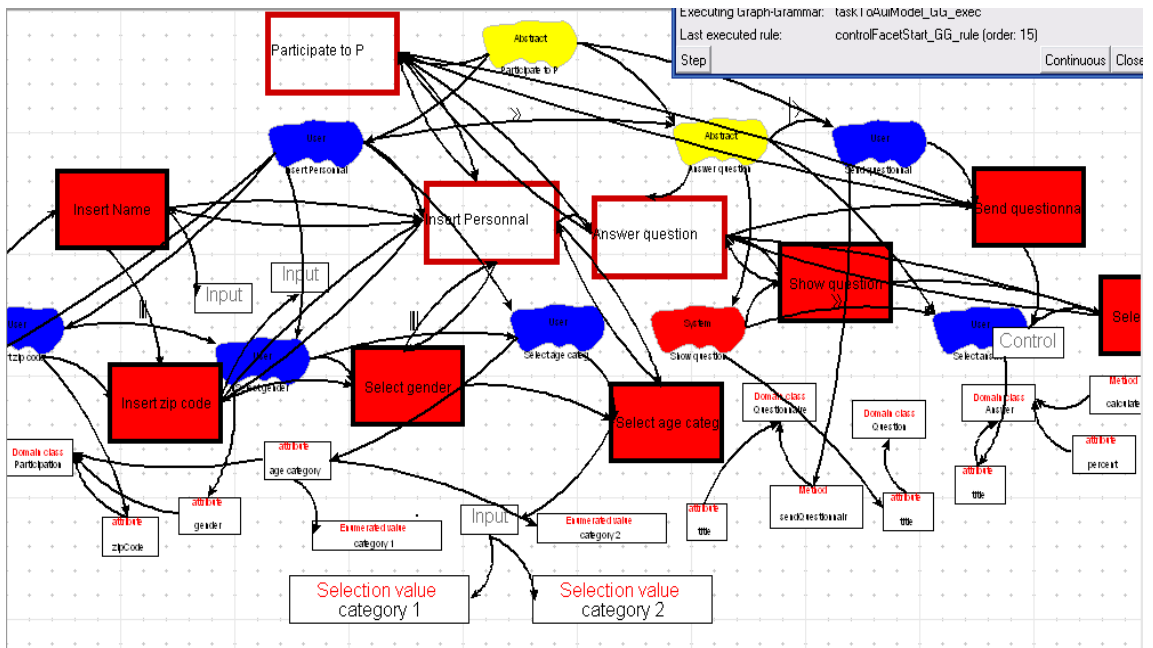


Figure 6-28. Virtual Polling System after execution of rule 10

On figure 6-28 shows that for each task of type “start” that manipulates a method, a control facet of type start has been created (we can see the control facet at the right side of the screenshot).

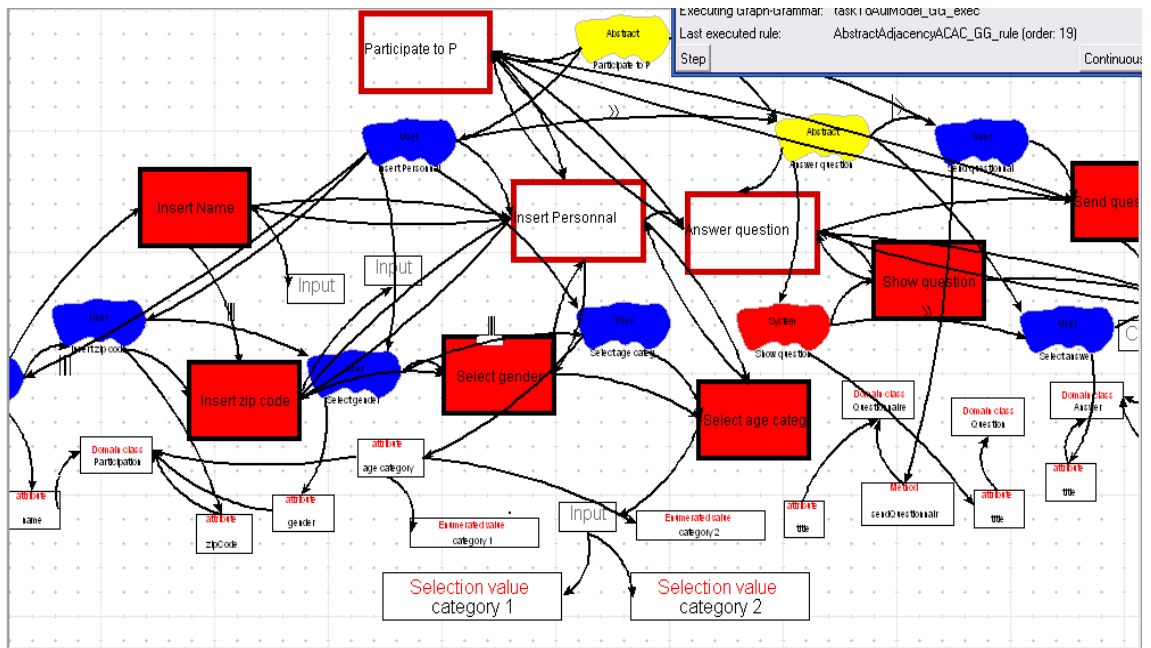


Figure 6-29. Virtual Polling System after execution of rule 11, 12, 13 and 14

We can't see it on the screenshot because ATOM³ make arrows override each other, but an *abstract adjacency* link has been created between each couple of sister tasks that are linked by an *enabling* link, which means that the two objects in the final user interface corresponding to those tasks will be next to each other (Figure 6-29).

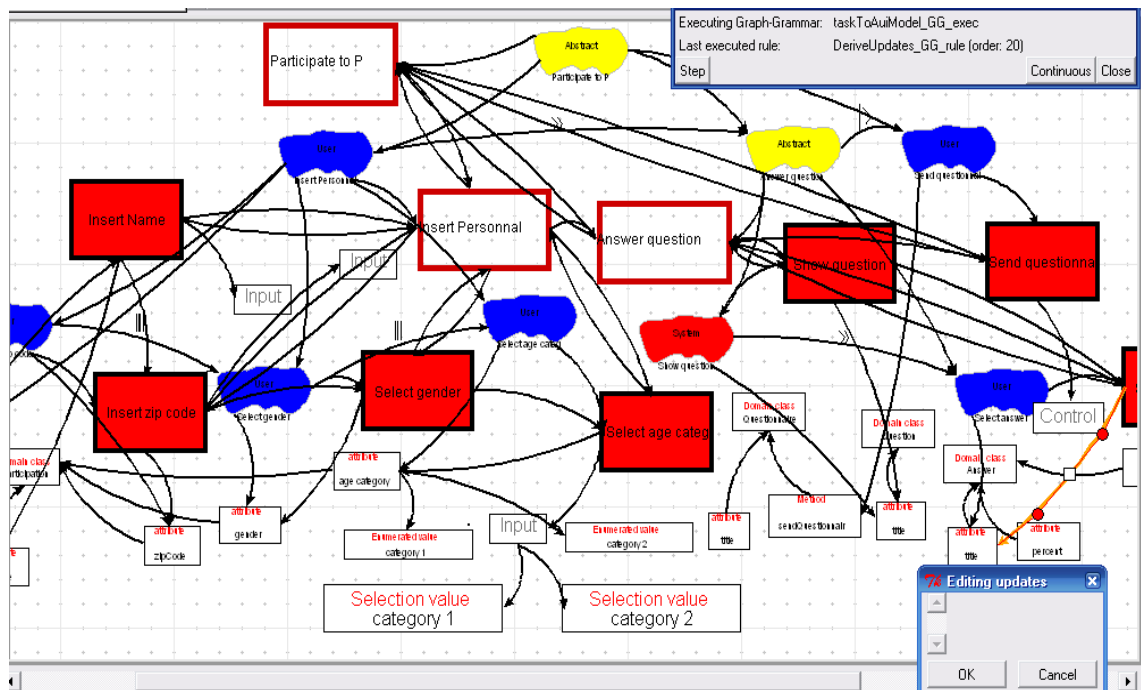


Figure 6-30. Virtual Polling System after execution of rule 19

For each task that manipulates an attribute, and that has been linked to an AIC, the AIC is link to the attribute with an *updates* relation (Figure 6-30).

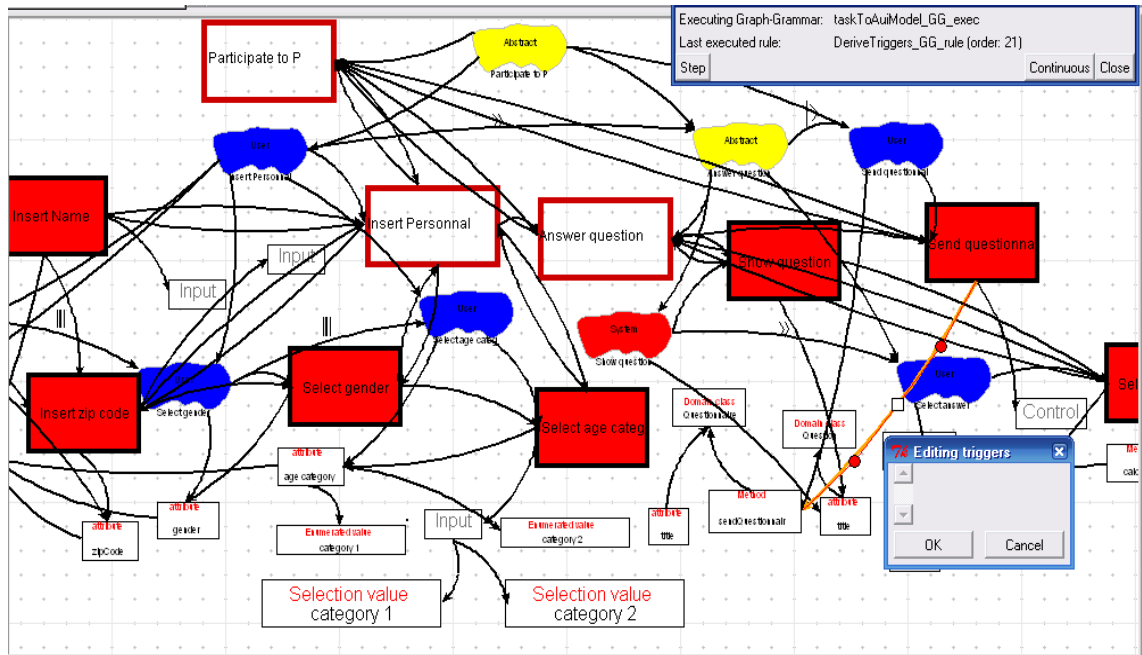


Figure 6-31. Virtual Polling System after execution of rule 20

For each task that manipulates a method, and that has been linked to an AIC, the AIC is linked to the attribute with a *triggers* relation (Figure 6-31).

For each task that manipulates an attribute, and that has been linked to an AIC, the AIC is link to the attribute with an *updates* relation.

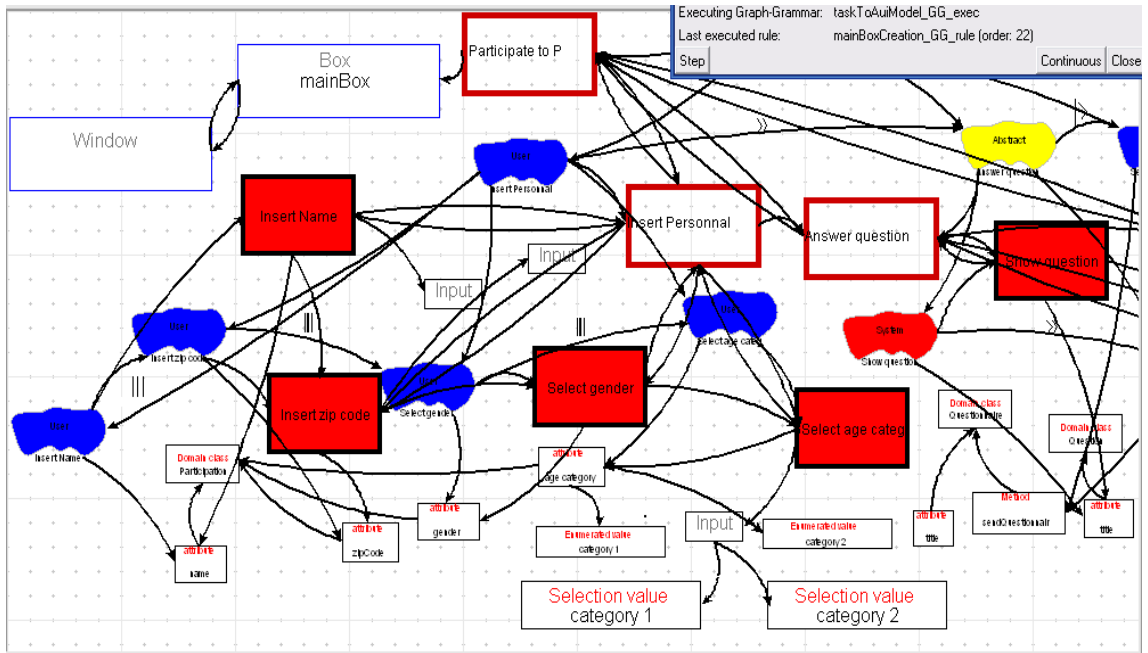


Figure 6-32. Virtual Polling System after execution of rule 21

Here, a *box* named *mainbox* (containing the *window* that will contain the entire graphical user interface) is created and linked to the root task. This will be the main *graphical container* (Figure 6-32).

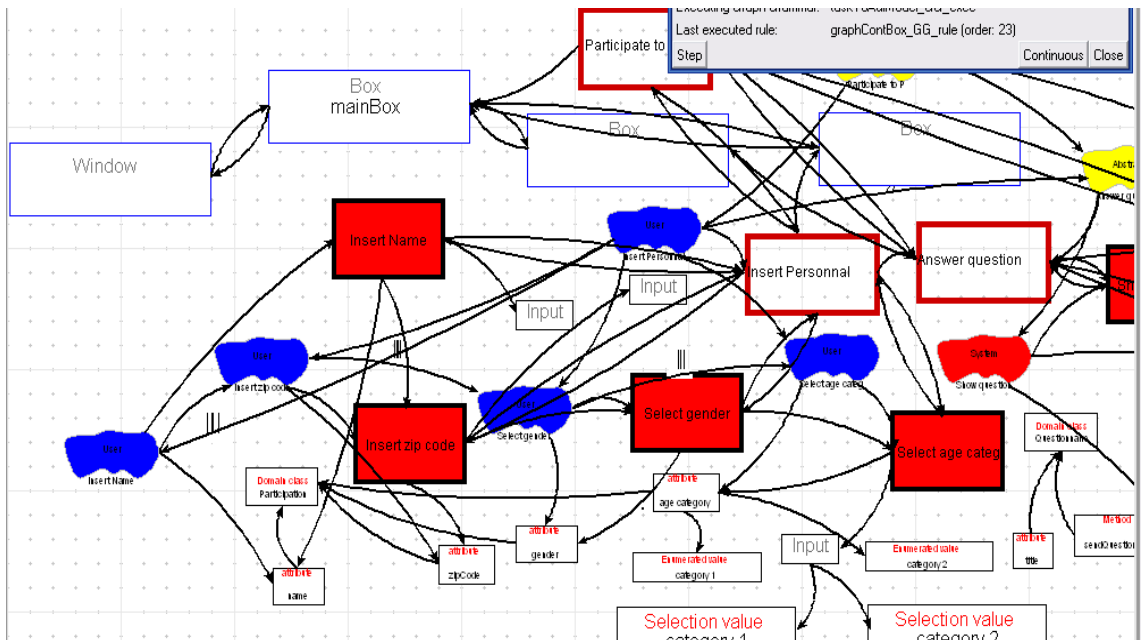


Figure 6-33 Virtual Polling System after execution of rule 22

On figure 6-33 we see that for each *abstract container*, a *graphical container* of type *box* is created, and linked to it (by a *reify* relation).

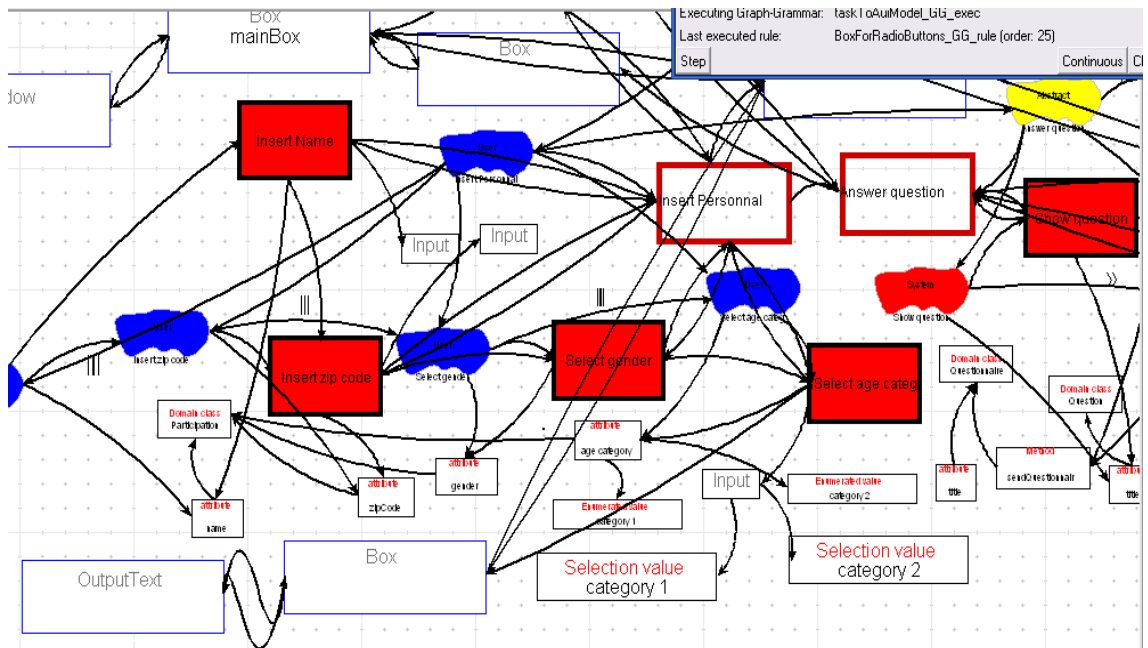


Figure 6-34. Virtual Polling System after execution of rule 24

Figure 6-34: A *box* is created that will contain the *radio buttons* corresponding to the *selection values* previously created. The *Box* also contains an *output text* graphical component, which will in fact contain a message telling the user what to do (“select an age category” for example).

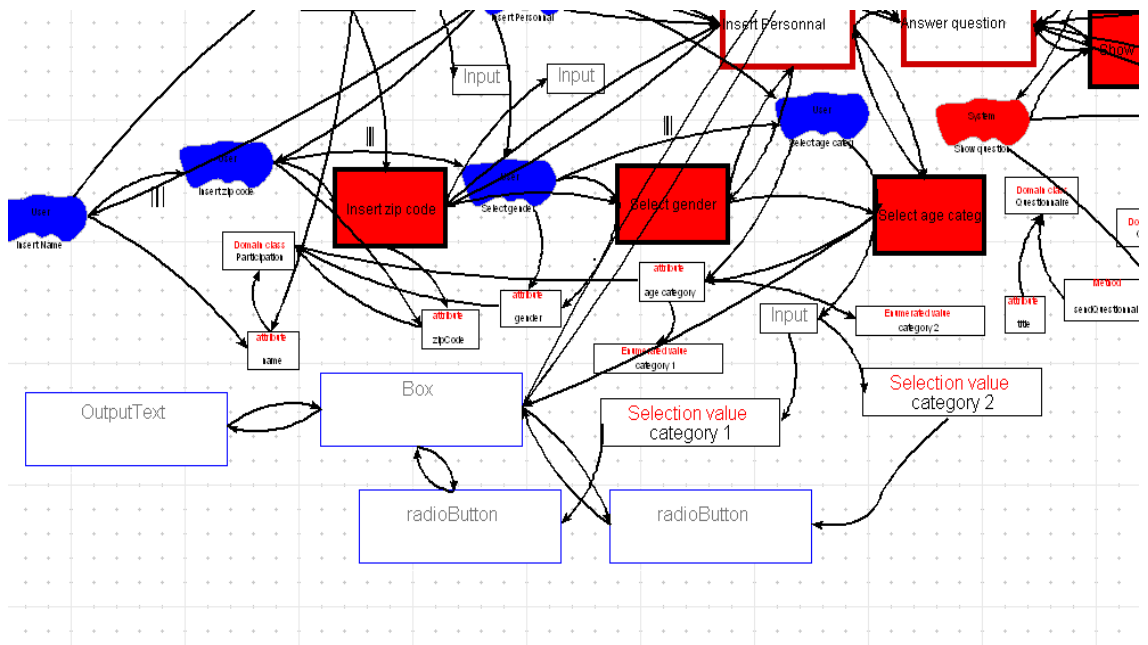


Figure 6-35. Virtual Polling System after execution of rule 25

For each *selection value*, a *radio button* is created, and contained by the *box* previously created for it (Figure 6-35).

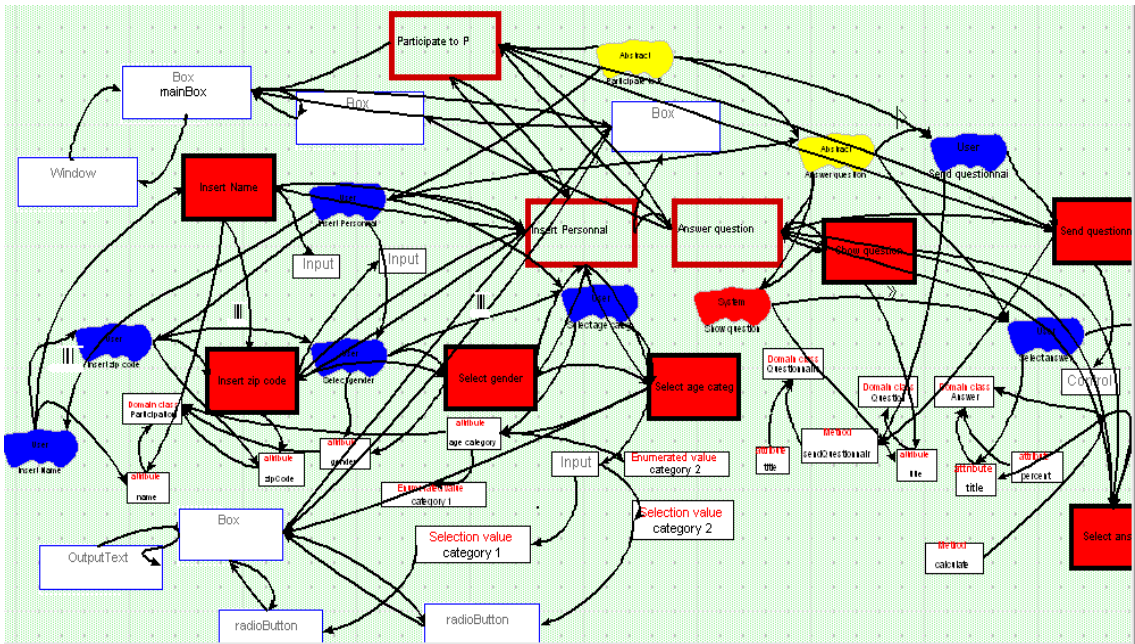


Figure 6-36. Virtual Polling System final result

Figure 6-36 shows the final result.

6.3.2 Custom java transformation engine

As for the previous example, we open the source file containing the virtual polling system model in UsiXML (see example description above). The result is the following (to gain space, we have withdrawn the task and domain model, since they aren't modified by the transformation engine):

```
<?xml version="1.0" encoding="UTF-8"?>
<uiModel id="UiM1" name="PollingSystem">

  <mappingModel>
    <manipulates id="MA1">
      <source sourceId="T11"/>
      <target targetId="A1DC1"/>
    </manipulates>
    <manipulates id="MA2">
      <source sourceId="T12"/>
      <target targetId="A2DC1"/>
    </manipulates>
    <manipulates id="MA3">
      <source sourceId="T13"/>
      <target targetId="A3DC1"/>
    </manipulates>
    <manipulates id="MA4">
      <source sourceId="T14"/>
      <target targetId="A4DC1"/>
    </manipulates>
  </mappingModel>
</uiModel>
```



```

<manipulates id="MA5">
  <source sourceId="T21"/>
  <target targetId="A1DC3"/>
</manipulates>
<manipulates id="MA6">
  <source sourceId="T22"/>
  <target targetId="A1DC4"/>
</manipulates>
<manipulates id="MA7">
  <source sourceId="T3"/>
  <target targetId="M1DC2"/>
</manipulates>
<updates>
  <source sourceId="AIC_T13"/>
  <target targetId="A3DC1"/>
</updates>
<updates>
  <source sourceId="AIC_T14"/>
  <target targetId="A4DC1"/>
</updates>
<updates>
  <source sourceId="AIC_T22"/>
  <target targetId="A1DC4"/>
</updates>
<triggers>
  <source sourceId="AIC_T3"/>
  <target targetId="M1DC2"/>
</triggers>
<isReifiedBy>
  <source sourceId="AC_Root"/>
  <target targetId="box_00"/>
</isReifiedBy>
<isReifiedBy>
  <source sourceId="AC_T1"/>
  <target targetId="Box_AC_T1"/>
</isReifiedBy>
<isReifiedBy>
  <source sourceId="AC_T2"/>
  <target targetId="Box_AC_T2"/>
</isReifiedBy>
<isReifiedBy>
  <source sourceId="AIC_T3"/>
  <target targetId="Box_AIC_T3"/>
</isReifiedBy>
<isReifiedBy>
  <source sourceId="AIC_T11"/>
  <target targetId="Box_create_AIC_T11"/>
</isReifiedBy>
<isReifiedBy>
  <source sourceId="AIC_T12"/>
  <target targetId="Box_create_AIC_T12"/>
</isReifiedBy>
<isReifiedBy>
  <source sourceId="AIC_T21"/>
  <target targetId="Box_create_AIC_T21"/>
</isReifiedBy>
<isReifiedBy>
  <source sourceId="AIC_T13"/>
  <target targetId="Box_Selection_AIC_T13"/>
</isReifiedBy>

```

```

<isReifiedBy>
  <source sourceId="AIC_T14"/>
  <target targetId="Box_Selection_AIC_T14"/>
</isReifiedBy>
<isReifiedBy>
  <source sourceId="AIC_T22"/>
  <target targetId="Box_Selection_AIC_T22"/>
</isReifiedBy>
<isReifiedBy>
  <source sourceId="AIC_T3"/>
  <target targetId="Button_FAT3"/>
</isReifiedBy>
<updates>
  <source sourceId="Box_Selection_AIC_T13"/>
  <target targetId="A3DC1"/>
</updates>
<updates>
  <source sourceId="Box_Selection_AIC_T14"/>
  <target targetId="A4DC1"/>
</updates>
<updates>
  <source sourceId="Box_Selection_AIC_T22"/>
  <target targetId="A1DC4"/>
</updates>
</mappingModel>
<guiModel>
  <abstractContainer name="Participate to poll" id="AC_Root">
    <abstractContainer name="Insert personal data" id="AC_T1">
      <abstractIndividualComponent name="Insert name"
        id="AIC_T11">
        <input id="FAT11" actionType="create"
          actionItem="element"/>
      </abstractIndividualComponent>
      <abstractIndividualComponent name="Insert zip code"
        id="AIC_T12">
        <input id="FAT12" actionType="create"
          actionItem="element"/>
      </abstractIndividualComponent>
      <abstractIndividualComponent name="Select gender"
        id="AIC_T13">
        <input id="FAT13" actionType="select"
          actionItem="element">
          <selectionValue name="Male"/>
          <selectionValue name="Female"/>
        </input>
      </abstractIndividualComponent>
      <abstractIndividualComponent name="Select age
        category" id="AIC_T14">
        <input id="FAT14" actionType="select"
          actionItem="element">
          <selectionValue name="18-35"/>
          <selectionValue name="35-45"/>
          <selectionValue name="45+"/>
        </input>
      </abstractIndividualComponent>
    </abstractContainer>
    <abstractContainer name="Answer question" id="AC_T2">
      <abstractIndividualComponent name="Show question"
        id="AIC_T21">
        <input id="FAT21" actionType="create"

```

```

        actionItem="element"/>
    </abstractIndividualComponent>
    <abstractIndividualComponent name="Select answer"
    id="AIC_T22">
        <input id="FAT22" actionType="select"
        actionItem="element">
            <selectionValue name="Question1"/>
            <selectionValue name="Question2"/>
        </input>
    </abstractIndividualComponent>
</abstractContainer>
<abstractIndividualComponent name="Send questionnaire"
id="AIC_T3">
    <control id="FAT3" actionType="start"
    actionItem="operation"/>
</abstractIndividualComponent>
</abstractContainer>
<abstractContainment id="ACont_AC_T1">
    <source sourceId="AC_T1"/>
    <target targetId="AIC_T11"/>
    <target targetId="AIC_T12"/>
    <target targetId="AIC_T13"/>
    <target targetId="AIC_T14"/>
</abstractContainment>
<abstractContainment id="ACont_AC_T2">
    <source sourceId="AC_T2"/>
    <target targetId="AIC_T21"/>
    <target targetId="AIC_T22"/>
</abstractContainment>
<abstractContainment id="ACont_AC_Root">
    <source sourceId="AC_Root"/>
    <target targetId="AC_T1"/>
    <target targetId="AC_T2"/>
    <target targetId="AIC_T3"/>
</abstractContainment>
<abstractAdjacency id="AA_AC_T1">
    <source sourceId="AC_T1"/>
    <target targetId="AC_T2"/>
</abstractAdjacency>
<abstractAdjacency id="AA_AIC_T21">
    <source sourceId="AIC_T21"/>
    <target targetId="AIC_T22"/>
</abstractAdjacency>
<auiDialogControl control="EnablingType" id="AuiDCnull">
    <source sourceId="AC_T1"/>
    <target targetId="AC_T2"/>
</auiDialogControl>
<auiDialogControl control="DisablingType" id="AuiDCnull">
    <source sourceId="AC_T2"/>
    <target targetId="AIC_T3"/>
</auiDialogControl>
<auiDialogControl control="IndependentConcurrencyType"
id="AuiDCnull">
    <source sourceId="AIC_T11"/>
    <target targetId="AIC_T12"/>
</auiDialogControl>
<auiDialogControl control="IndependentConcurrencyType"
id="AuiDCnull">
    <source sourceId="AIC_T12"/>
    <target targetId="AIC_T13"/>

```

```

</audiDialogControl>
<audiDialogControl control="IndependentConcurrencyType"
  id="AuiDCnull">
  <source sourceId="AIC_T13"/>
  <target targetId="AIC_T14"/>
</audiDialogControl>
<audiDialogControl control="EnablingType" id="AuiDCnull">
  <source sourceId="AIC_T21"/>
  <target targetId="AIC_T22"/>
</audiDialogControl>
</audiModel>
<cuiModel>
  <box name="main_box" id="box_00"/>
  <window id="window_00"/>
  <box id="Box_AC_T1"/>
  <box id="Box_AC_T2"/>
  <box id="Box_AIC_T3"/>
  <box id="Box_create_AIC_T11"/>
  <inputText textSize="12" id="Input_text_FAT11"
    textColor="#000000"/>
  <outputText textSize="12" id="Output_text_FAT11"
    textColor="#000000"/>
  <box id="Box_create_AIC_T12"/>
  <inputText textSize="12" id="Input_text_FAT12"
    textColor="#000000"/>
  <outputText textSize="12" id="Output_text_FAT12"
    textColor="#000000"/>
  <box id="Box_create_AIC_T21"/>
  <inputText textSize="12" id="Input_text_FAT21"
    textColor="#000000"/>
  <outputText textSize="12" id="Output_text_FAT21"
    textColor="#000000"/>
  <box id="Box_Selection_AIC_T13"/>
  <outputText textSize="12" id="Output_text_FAT13"
    textColor="#000000">
    <defaultContent>Select gender</defaultContent>
  </outputText>
  <box id="Box_Selection_AIC_T14"/>
  <outputText textSize="12" id="Output_text_FAT14"
    textColor="#000000">
    <defaultContent>Select age category</defaultContent>
  </outputText>
  <box id="Box_Selection_AIC_T22"/>
  <outputText textSize="12" id="Output_text_FAT22"
    textColor="#000000">
    <defaultContent>Select answer</defaultContent>
  </outputText>
  <radioButton id="radioButton_FAT14_1" groupName="Select age
    category">
    <textSize>12</textSize>
    <defaultContent>18-35</defaultContent>
    <textColor>#000000</textColor>
  </radioButton>
  <radioButton id="radioButton_FAT14_2" groupName="Select age
    category">
    <textSize>12</textSize>
    <defaultContent>35-45</defaultContent>
    <textColor>#000000</textColor>
  </radioButton>
  <radioButton id="radioButton_FAT14_3" groupName="Select age

```

```

category">
    <textSize>12</textSize>
    <defaultContent>45+</defaultContent>
    <textColor>#000000</textColor>
</radioButton>
<radioButton id="radioButton_FAT22_1" groupName="Select
answer">
    <textSize>12</textSize>
    <defaultContent>Question1</defaultContent>
    <textColor>#000000</textColor>
</radioButton>
<radioButton id="radioButton_FAT22_2" groupName="Select
answer">
    <textSize>12</textSize>
    <defaultContent>Question2</defaultContent>
    <textColor>#000000</textColor>
</radioButton>
<button id="Button_FAT3">
    <textSize>12</textSize>
    <textColor>#000000</textColor>
</button>
<graphicalContainment>
    <source sourceId="box_00"/>
    <target targetId="window_00"/>
</graphicalContainment>
<graphicalContainment>
    <source sourceId="box_00"/>
    <target targetId="Box_AC_T1"/>
</graphicalContainment>
<graphicalContainment>
    <source sourceId="box_00"/>
    <target targetId="Box_AC_T2"/>
</graphicalContainment>
<graphicalContainment>
    <source sourceId="box_00"/>
    <target targetId="Box_AIC_T3"/>
</graphicalContainment>
<graphicalContainment>
    <source sourceId="Box_create_AIC_T11"/>
    <target targetId="Output_text_FAT11"/>
</graphicalContainment>
<graphicalContainment>
    <source sourceId="Box_create_AIC_T11"/>
    <target targetId="Input_text_FAT11"/>
</graphicalContainment>
<graphicalAdjacency>
    <source sourceId="Input_text_FAT11"/>
    <target targetId="Output_text_FAT11"/>
</graphicalAdjacency>
<graphicalContainment>
    <source sourceId="Box_create_AIC_T12"/>
    <target targetId="Output_text_FAT12"/>
</graphicalContainment>
<graphicalContainment>
    <source sourceId="Box_create_AIC_T12"/>
    <target targetId="Input_text_FAT12"/>
</graphicalContainment>
<graphicalAdjacency>
    <source sourceId="Input_text_FAT12"/>
    <target targetId="Output_text_FAT12"/>

```

```

</graphicalAdjacency>
<graphicalContainment>
  <source sourceId="Box_create_AIC_T21"/>
  <target targetId="Output_text_FAT21"/>
</graphicalContainment>
<graphicalContainment>
  <source sourceId="Box_create_AIC_T21"/>
  <target targetId="Input_text_FAT21"/>
</graphicalContainment>
<graphicalAdjacency>
  <source sourceId="Input_text_FAT21"/>
  <target targetId="Output_text_FAT21"/>
</graphicalAdjacency>
<graphicalContainment>
  <source sourceId="Box_Selection_AIC_T13"/>
  <target targetId="Output_text_FAT13"/>
</graphicalContainment>
<graphicalContainment>
  <source sourceId="Box_AC_T1"/>
  <target targetId="Box_Selection_AIC_T13"/>
</graphicalContainment>
<graphicalContainment>
  <source sourceId="Box_Selection_AIC_T14"/>
  <target targetId="Output_text_FAT14"/>
</graphicalContainment>
<graphicalContainment>
  <source sourceId="Box_AC_T1"/>
  <target targetId="Box_Selection_AIC_T14"/>
</graphicalContainment>
<graphicalContainment>
  <source sourceId="Box_Selection_AIC_T22"/>
  <target targetId="Output_text_FAT22"/>
</graphicalContainment>
<graphicalContainment>
  <source sourceId="Box_AC_T2"/>
  <target targetId="Box_Selection_AIC_T22"/>
</graphicalContainment>
<graphicalContainment>
  <source sourceId="Box_Selection_AIC_T14"/>
  <target targetId="radioButton_FAT14_1"/>
</graphicalContainment>
<graphicalContainment>
  <source sourceId="Box_Selection_AIC_T14"/>
  <target targetId="radioButton_FAT14_2"/>
</graphicalContainment>
<graphicalContainment>
  <source sourceId="Box_Selection_AIC_T14"/>
  <target targetId="radioButton_FAT14_3"/>
</graphicalContainment>
<graphicalContainment>
  <source sourceId="Box_Selection_AIC_T22"/>
  <target targetId="radioButton_FAT22_1"/>
</graphicalContainment>
<graphicalContainment>
  <source sourceId="Box_Selection_AIC_T22"/>
  <target targetId="radioButton_FAT22_2"/>
</graphicalContainment>
<graphicalContainment>
  <source sourceId="box_00"/>
  <target targetId="Button_FAT3"/>

```

```

</graphicalContainment>
<graphicalAdjacency>
  <source sourceId="Box_AC_T1"/>
  <target targetId="Box_AC_T2"/>
</graphicalAdjacency>
<graphicalAdjacency>
  <source sourceId="Box_create_AIC_T21"/>
  <target targetId="Box_Selection_AIC_T22"/>
</graphicalAdjacency>
<cuiDialogControl>
  <symbol>EnablingType</symbol>
  <source sourceId="Box_AC_T1"/>
  <target targetId="Box_AC_T2"/>
</cuiDialogControl>
<cuiDialogControl>
  <symbol>DisablingType</symbol>
  <source sourceId="Box_AC_T2"/>
  <target targetId="Box_AIC_T3"/>
</cuiDialogControl>
<cuiDialogControl>
  <symbol>DisablingType</symbol>
  <source sourceId="Box_AC_T2"/>
  <target targetId="Button_FAT3"/>
</cuiDialogControl>
<cuiDialogControl>
  <symbol>IndependentConcurrencyType</symbol>
  <source sourceId="Box_create_AIC_T11"/>
  <target targetId="Box_create_AIC_T12"/>
</cuiDialogControl>
<cuiDialogControl>
  <symbol>IndependentConcurrencyType</symbol>
  <source sourceId="Box_create_AIC_T12"/>
  <target targetId="Box_Selection_AIC_T13"/>
</cuiDialogControl>
<cuiDialogControl>
  <symbol>IndependentConcurrencyType</symbol>
  <source sourceId="Box_Selection_AIC_T13"/>
  <target targetId="Box_Selection_AIC_T14"/>
</cuiDialogControl>
<cuiDialogControl>
  <symbol>EnablingType</symbol>
  <source sourceId="Box_create_AIC_T21"/>
  <target targetId="Box_Selection_AIC_T22"/>
</cuiDialogControl>
</cuiModel>
</uiModel>

```

Figure 6-37. Result of the virtual polling system

As we can see, all the rules of the rule set that are relevant for the example (see example description) have been executed.

Chapter 7: Comparison of the four techniques

First, we will give a short summary of each of the three techniques. Then we will make a global comparison, using criterion taken from the work of [mister x]. ... Thirdly, we will resume each transformation engine, reminding the specific issues with each of them. Finally, we will make a short conclusion based on what we experienced with each tool, and what needs are fulfilled by each of them.

7.1 Summaries

7.1.1 AToM³

AToM³ is graph transformation tool. It means that the meta-model (here, the one of UsiXML) has to be graphically designed, as well as the transformations we want to implement. The only thing that is implemented using a programming language (actually Python), are pre- and post-conditions for the transformations, as well as some things that can't be graphically specified, such as changing the name of a variable, or specifying a variable's value basing on other variables.

The structure of UsiXML is easily represented as an oriented graph, which means that AToM³ suits well the needs of this language.

Graph transformation rules consist of a Left Hand Side (LHS) matching a graph G, a Right Hand Side that has to be the result of the transformation of the graph G, and pre- and post-conditions, coded in the Python programming language. The graphical representation of the rules is very intuitive, and only small things have to be coded in Python, so the GUI designer doesn't have to be highly skilled in programming languages.

From the point of view of the time needed to implement everything, though, AToM³ is not as good as we might think, for some reasons:

- First, the program itself is coded in Python, which means that it is quite slow, and the GUI designer loses some time during graph layout processing and some other things. This is not a major matter, though.
- The way of creating the rules, with one window for each rule, plus two windows for the pre-condition and the post-action, also means a small loss of time, compared to programming in a single environment, like we would do for a java project for example.
- More importantly, in AToM³ more transformation rules have to be coded than in a wholly java-programmed project for example. The reason is simple: suppose we want a rule that apply on each couple of task linked by a relation, no matter the type of the relation. In AToM³, if there exist different relations that do not extend the same "parent" relation (by inheritance), we have to create one rule per type of

relation. In java, at the other hand, we can simply do it in a single method, with something like (in pseudo-code): `if(exists(relation) & relation.source = A & relation.target = B) { ... }`

This means an important loss of time, and an increased complexity.

- Finally, AToM³ is still in development, which means it has some problems of stability that can also make GUI designer lose time.

But, despite all those negative points, it makes no doubt that the graphical approach of AToM³ is the simplest and more intuitive way of creating transformation rules.

7.1.2 ATL

ATL is some kind of “hybrid” approach: it uses graphs to implement the meta-model, but the user can use the ATL specific language to design the transformation rules.

For the same reason as for AToM³, the graph representation of the meta-model is particularly well suited to UsiXML.

7.1.3 Custom transformation engine in java

Here, everything is coded “by hand” in the java programming language. That means that, from the reading of the UsiXML file, to the writing of the result in UsiXML, everything is coded in java. The UsiXML file is read with the *unmarshaller* of the *Castor* api. It is transformed in java objects that can thus be manipulated through java methods.

Castor needs a mapping file, which contains the mapping between the XML and the java objects representation of the file. This mapping file is created from the xsd schema of UsiXML.

After the UsiXML file has been read, and transformations have been applied to the internal representation (java objects) of the file, the internal representation is translated back into UsiXML, using the mapping file to respect the syntax of the UsiXML language.

7.2 Comparison

7.2.1 Tools-specific issues

7.2.1.1 AToM³

7.2.1.1.1 Meta-model and models creation

The meta-model and all models have to be created using a UML-like diagram, in an add-hoc canvas. However, this raises two issues:

- The UML relations are restricted in AToM³: there is no composition relation. To create a composition relation, we have to create ourselves a bidirectional relation between the two objects. What's more, AToM³ cannot handle two relations having the same name, so each "fake" composition relation must have a name different from the others.
- The canvas is too small: big models are hard to design in Atom³. To do this, we have to use the zoom to reduce it and be able to add objects. But this shows another problem of AToM³: it is not perfectly stable; using the zoom can lead to unexpected things like objects cross (or cover) each others, which further reduce the already poor readability of the models in AToM³.

These two issues mean that AToM³ isn't as fast as expected to create big projects.

7.2.1.1.1 Rules creation

Rules are created in AToM³ using the window showed in figure 3-5. This also raises some issues:

- The canvas to create the LHS and RHS patterns are very small, which can lead to room problems.
- Closing a rule, then opening it again can lead to problems with indexes of the objects : trying to add a new one in the LHS or RHS pattern will most of the time create an object with index "1" (it starts back from 0), which we can change after. But deleting an object in an open-then-closed rule, and then creating a new one with the same index will cause an "interference": AToM³ then thinks that two objects have the same index (which is not true). So in general, to avoid problems we have to create a rule in one time (this is not always easy, and reduce the easiness maintenance of our transformation engine).
- Condition and action, as well as algorithms in python to compute the value of variables are all created in separate windows. Because the program itself (written in python) is quite slow, the windows

can take some times to open, and most of the time the windows that opens is too small and we have to resize it. While not being a severe problem, this makes the rule long to create because we have to use many different windows, even for a very small rule.

- There is no inheritance mechanism for the rules.

7.2.1.2 Custom java transformation engine

7.2.1.2.1 Meta-model creation

- Our tool has no internal mechanism to create a meta-model. In our case, this has been done with [jaxb]. Taking an .xsd schema of the meta-model, it creates java objects for each met entity, and accessors for each attribute of these objects. But, the process is not easy :
 - The user has to be able to use jaxb
 - An .xsd schema of the meta-model must obligatorily be available.
- The program needs, to be able to read and write XML files, a mapping file. With a meta-model like UsiXML, this mapping is very long (more than 10000 lines). There exist tools to automatically create it, but they don't work perfectly: because the tool we used [tool] was not able to create "array list" items in the mapping file, we had to modify manually the mapping file to add all the lists. This is long, and forces us to learn the syntax of the mapping file.
- Because of precedent, the application is not really flexible: ours is designed for only one meta-model, and creating another means long and hard work.

7.2.1.2.2 Model creation

- There isn't either an internal tool to create models in our java transformation engine. The only way to create a model in our application is by writing it directly in the source text area. This is not easy, because it offers no correction facility or syntax highlighting. However, there exist external tools like idealXML [idealXML] that are very intuitive and easy to use to create models in UsiXML. The lack of an internal tool of our application is thus not really a problem.

7.2.1.2.3 Rules creation

- Rules are created programmatically, which means programming skills are mandatory.
- Optimization is now up to the programmer if he wants his engine not to be too slow.
- Creating a rule not only means coding it, but also modifying the rules tree (at the right of the figure 4-3) and calling it in the main program.

7.2.2 Tools-specific advantages

7.2.2.1 AToM³

- Very fast for small projects: designing small models in AToM³ is very fast because of its graphical syntax.
- Easy pattern representation: the graphical syntax is intuitive and easy to use and understand.
- Python programming and templates: conditions, actions and computing are easy to implement because of the easy syntax of Python and, what's more, templates are present in AToM³ to help the user.
- Automatically generated interface for model creation: once a meta-model is created, AToM³ generate an interface to create models conforming to the meta-model, and transformation rules using the concepts of the meta-model.
- Cardinalities verification: when creating or modifying a relation in a model, AToM³ checks that the cardinalities (number of sources and targets) are respected. So, the meta-model cannot be violated in a model.
- Step-by-step execution: the transformation rules can be executed one at a time, which can help finding the ones that are not working.

7.2.2.2 Custom java transformation engine

- Completeness of the transformation engine: at the price of a harder syntax than in AToM³, any kind of transformations can be implemented, and there are no limitations in the creation of meta-models.
- Explicit flow control: because transformation rules are methods, any transformation rule can call any other.
- Code generation: our transformation engine generates UsiXML files only but any kind of XML-based code can be generated using the Castor project as we did and even any kind of code can be generated but at the cost of a harder implementation.
- The interface can be adopted by the user (at the price of modifying the interface source code) to meet his needs.
- Because our custom transformation engine takes UsiXML files as input, any external tool can be used to create models, like [idealXML].
- Numerous existing java technologies: we didn't use them, but many technologies exist in java (like JMI) to help designing a custom transformation engine.
- Better error codes: error codes in java are easier to understand. This allows the user to find more easily not working transformation rules. What's more, the user can take profit of facilities offered by the IDE used (such as Eclipse), like the debugger, error checking, and so on.

7.2.3 Global comparison

We won't compare here the tools regarding the implementation details of each, or the problems we had using it. The point 7.2.1 was devoted to it. We will instead compare the concepts promoted by each one. For this, we will use seven criteria inspired by the work of [Czarnecki] and [Schaefer]:

- Implementation paradigm, and the need of programming skills
- Model-to-model approach
- Code generation
- Pattern matching
- Rules ordering and Rules organization

- (Bi-)Directionality
- Performance
- Flexibility, maintainability

7.2.3.1 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 is strictly declarative; the java application is in imperative style, and ATL can use both paradigms.

AToM³ needs almost no programming skills. There are a few things to code in python, like conditions, actions, constraints and variables' valuing, but these are coded in a very simple syntax (the python language itself is quite simple), and the syntax is most of the time explained in AToM³.

ATL needs higher programming skills, because the transformation rules use OCL syntax. But the GUI designer is helped in ATL by several facilities provided: the meta-model can be created graphically, and there is a pattern matching instruction. What's more, the syntax is far easier than the syntax of a rule coded in java.

The easiness of AToM³ and ATL makes them more flexible, because even GUI designers with (almost) no programming skills can use it, and also because maintaining the rules is much simplified by the fact that the representation of those is clear and easy to understand. However, in both AToM³ and ATL, the transformation rules as specified specifically for the meta-model that we have graphically created, and a change to this meta-model will make the transformations not work anymore.

Java seems thus a worse approach to the creation of the transformations here, especially for non-programmers, while ATL looks more like an "in-between" solution, suitable for people with few programming skills.

7.2.3.2 The model-to-model approach

AToM³ uses graph transformation. The models are created graphically, and so are the transformation rules. The latter are composed of two parts: a LHS graph pattern, and a RHS graph pattern. The LHS graph pattern will be searched in the source graph, and replaced by the RHS. This is a very intuitive, and easy to read and maintain, way of creating rules.

The java application 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 our java application, a method is created for each transformation rule. As this is code, this is much harder to read and maintain.

The ATL approach is some kind of hybrid: the meta-model is created graphically (it can also be created programmatically), and the transformation rules are coded. These ones can be coded declaratively or imperatively. If keeping with the declarative approach, the code is still simple to read and maintain, even if it requires a bit more programming knowledge than AToM³.

7.2.3.3 Code generation

In java, code generation is straightforward and simple. We used the Castor api for this purpose, and it causes no major problems.

ATL can also generate code, with help of an external model handler.

Finally, AToM³ hasn't any commodity to generate code, nor reading it. 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.

7.2.3.4 Pattern matching

This is a big difference between AToM³, ATL and our java application. While the pattern matching is obviously supported by AToM³, and given with a simple syntax in ATL, java doesn't provide it. Pattern matching is still usable in java, but with the help of an existing external project, or at the price of a long and fastidious implementation. This is much more complex to implement than in AToM³ and ATL, and we didn't use it. This also makes the java approach more complex than the two others.

7.2.3.5 Rules scheduling and organization, inheritance

AToM³ only allows determining a fixed order on the rules. There is no possibility of explicit flow control in AToM³. And there isn't a "call" instruction for a rule to call another. In fact, the precondition allows deciding whether or not the rule will be executed, but if it's not executed in its turn, then the rule will never be, so the precondition cannot be used for explicit flow control.

In AToM³, we can create distinct rules sets, and execute one after another. But there is no rule inheritance mechanism, and no import mechanism either: this means that a rules set cannot import a rule of another set. And a rule can neither use another.

On the opposite, ATL allows both implicit and explicit flow control. In purely declarative rules, the flow control is implicit, and non deterministic. But we can add imperative code in a rule to call others rules, and thus make explicit flow control. ATL even allows calling external, native code.

Java, finally, obviously allows rules organization (for example one class for each rules set), and the flow control is of course explicit. We can also call a rule in another, because each rule is a method in java.

7.2.3.6 Bi-directionality

None of the three supports bi-directionality, which means they are not suited for reverse engineering.

7.2.3.7 Performance

Here shows AToM³ his big disadvantage: it is very slow. AToM³ is entirely coded in Python, and compiled at execution. It means that the execution is much slower than, for instance, code compiled in java. Furthermore, the fact that the transformations are graphically designed means that the user has no possibility to optimize how they are executed. It's up to AToM³ to do it. And finally, because each transformation rule is created independently of the others, the pattern matching for the LHS will be executed for each rule, even if two of them have the same LHS.

Java is much more convenient to make something fast at the execution. Not only because the compiled code is faster than the interpreted code of AToM³, but also and mainly because here the programmer can choose himself exactly how the transformation will be executed, to find the faster execution. He can also for example eliminate redundancy by grouping transformation rules that have code in common, preventing the program from doing it twice.

But, once again, we have to make a choice between performance and flexibility: choosing to optimize the java code by grouping transformations means that the code will be harder to read and maintain. That's the reason why we chose to use one method per transformation rule in our project, even if this means a loss in terms of performance.

As we didn't use ATL ourselves, we cannot measure its performance, so this criterion will be left empty for ATL.

7.2.3.8 Flexibility and maintainability

The most readable the models, the easiest to maintain they are. And it also means they are easier to modify, thus making the tool the more flexible.

According to this, AToM³ offers the best to modify and maintains meta-models, models and transformation rules. However, two issues moderate that a little:

- First, and that is the case for each tool we used, modifying a meta-model means that the transformation rules designed for that meta-model are potentially not working anymore (because they can apply on objects that do not exist anymore, or have been modified).

- Second, modifying rules is sometimes difficult because of a problem of AToM³: modifying a previously saved rule in AToM³ sometimes leads to unexpected behavior (cf. chapter three).

ATL, because of its slightly more complex syntax for the transformation rules (meta-models are graphical), may be less readable and a little more difficult to maintain.

Finally, because of its fully programmatic approach and more complex syntax, the java application is the most difficult to maintain. And modifying it is very difficult because it implies several modifications in several classes in addition to coding the transformation rule. So, flexibility is worse than for the two other tools.

7.2.3.9 Completeness

With “completeness, we mean ability to handle complex rules and generate code. Internal tools for the creation of meta-models and models are not part of this criterion.

Because AToM³ allows doing many things in python, it is able of executing 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.

ATL can handle more complex rules than AToM³ because of its imperative constructs and explicit flow control (in the imperative blocks), but there are still limitations: the source model can be navigated, but not modified and the target model cannot even be navigated. This means we cannot make rules depending on the result of others.

Finally, in java, the limit is in fact the programming skills of the graphical interface designer.

Figure 7.1 summarizes all these criterion.

	AToM³	ATL	Java
Implementation paradigm	Declarative	Declarative and Imperative	Imperative
Programming skills needed, learning time	Few, short learning period	Slightly more than AToM³, medium learning period	High, long learning period
Model-to-model approach	Graph transformation	Direct model transformation	Hybrid
Pattern matching	Supported	Supported	Supported via external tools
Code generation	Not supported	Supported through external model handlers	Natively supported
Rules scheduling	Fixed order, no dynamic flow control	Scheduling by ATL for declarative rules, explicit flow control for imperative constructs.	Explicit flow control
Rules organization	Distinct rules sets, no inheritance	Distinct rules sets, inheritance supported	No limitations
Bi-directionality	Not supported	Not supported	Not supported

Figure 7-1 Tools classification

The last four criteria showed in the figure 7.2, namely completeness, performance, flexibility and maintainability are scored from very poor to very good: (thus very poor, poor, average, good or very good)

	Completeness	Performance	Flexibility	Maintainability
AToM³	Average	Poor	Very good	Good
Java	Very good	Good*	Very poor	Very poor
ATL	Good	-	Good	Very good

Figure 7-2 Tools classification (part 2)

* Depending on how the user optimize the code.

7.3.3 Conclusion

In the next graph, we merge completeness and performance on the x axis, and flexibility and maintainability on the y axis.

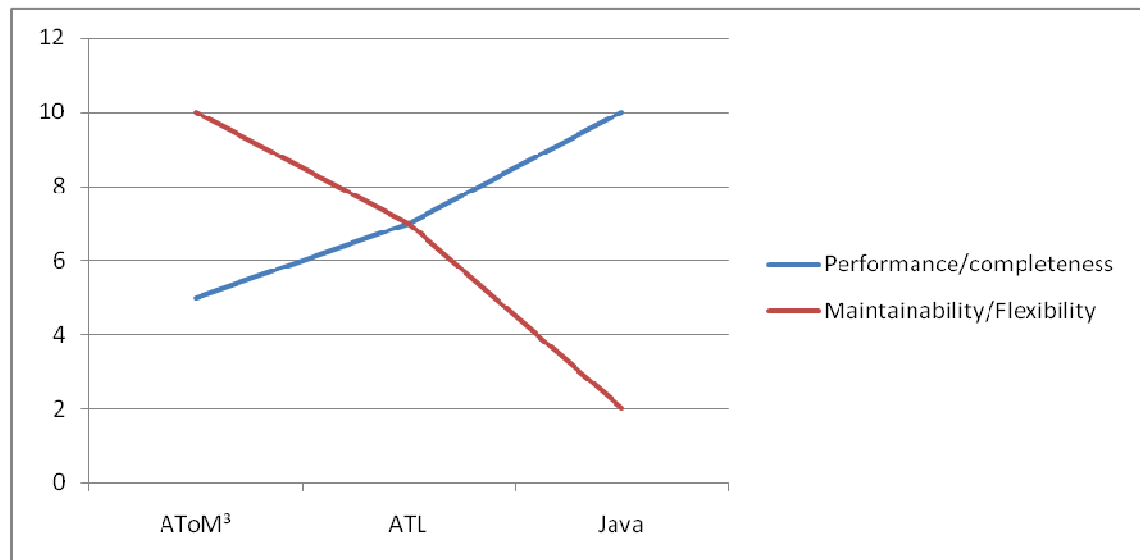


Figure 7-3 Tools comparison

As we can see on figure 7.3, no solution is simply better than the others, because our preference will depend on our needs.

AToM³ is certainly by far the simplest and easiest solution for model-to-model transformation, and it is also the most flexible. But it is unable of code generation, so it is unable of XML-transformation.

ATL is slightly harder to use than AToM³, because it requires higher programming skills. But ATL is more “powerful” than AToM³. First, while AToM³ only allows giving a fixed order on the rules, ATL allows implicit or explicit scheduling. Explicit scheduling is possible in ATL because of its rules inheritance support, which AToM³ doesn’t have.

What’s more, ATL supports both declarative and imperative programming, while AToM³ is strictly declarative. Imperative programming, while being harder to use and potentially less optimized, is useful for more complex rules (for example, it allows flow control with “if” and “call” instructions).

Java is certainly the hardest solution to implement, but is also offers the widest possibilities of the three “tools” we used.

In java, rules scheduling and organization is of course possible. Because of java’s imperative style and the [absence] of limitations [propres à un outil], the GUI designer can implement almost any kind of transformation on the model. But this has a [revers de la médaille]:

- Rules ordering is now up to the programmer, which is not [forcément] good in terms of optimization.
- The program is not guaranteed anymore to be deterministic (because Source and Target models are modifiable).
- Termination is not guaranteed anymore either.
- Readability is poor and,
- The most important: maintainability is very complex, mainly because of the lack of readability.

However, in java we can make complex rules, and code generation is easy to implement (we implemented it with the castor project).

What we said shows that, while being the most complete and powerful solution, the java application we wrote isn't really usable. But it would have been different if we had used pattern matching.

Using pattern matching means that the java application reads a model in (Usi)XML, transform it into java objects, and then do the same with the transformations also in (Usi)XML. The application then search (by pattern matching) the LHS pattern in the source model and replaces it by the RHS pattern. Finally, the application writes back the modified model into (Usi)XML.

With this solution, we never modify the java application to add rules or modify ones. It is then much more readable and easy to maintain. But this solution also means that we do not use the imperative possibilities of java. So the java application is less powerful with pattern matching than without it, but much more usable and maintainable, much more flexible.

The conclusion is then that, inevitably, more flexibility and maintainability means, for the GUI designer, more sacrifices to accept in terms of completeness (and functionality) of the application.

But if we need to write very complex rules, and need for example an explicit flow control, AToM³ is insufficient, and harder to use tools are mandatory, with the implementation time it implies.

Bibliography

[OMG] Official site of the Object Management Group, omg.org

[MDA] Official site of the OMG Model Driven Architecture, www.omg.org/mda

[1] K. Czarnecki and S. Helsen. [Feature-Based Survey of Model Transformation Approaches](#). IBM Systems Journal, special issue on Model-Driven Software Development. 45(3), 2006, pp. 621-645

[Czarnecki] K. Czarnecki and S. Helsen. [Classification of Model Transformation Approaches](#). In online proceedings of the 2nd OOPSLA'03 Workshop on Generative Techniques in the Context of MDA. Anaheim, October, 2003

[Stanciulescu] Adrian Stanciulescu, *A Methodology for Developing Multimodal User Interfaces of Information Systems*, 2007

[4] MDA journal, may 2004, <http://www.bptrends.com/publicationfiles/05-04%20COL%20IBM%20Manifesto%20-%20Frankel%20-3.pdf>

[Favre] Jean-Marie Favre, *Meta-Model and Model Co-evolution within the 3D Software Space*, 2004

[GreAT] A. Agrawal, G. Karsai and F. Shi., *Graph Transformations on Domain-Specific Models*. Under consideration for publication in the Journal on Software and Systems Modeling, 2003

[UMLX] E. D. Willink, *UMLX: A graphical transformation language for MDA*. In [Ren03], pp. 13-24

[VIATRA] D. Varro, G. Varro and A. Pataricza, *Designing the automatic transformation of visual languages*. Science of Computer Programming, vol. 44(2):pp. 205--227, 2002.

[BOTL] P. Braun and F. Marschall, *The Bi-directional Object-Oriented Transformation Language*. Technical Report, Technische Universität München, TUM-I0307, May 2003

[Nauwenko] Andrey Naumenko, Alain Wegmann, *two approaches in system modeling and their illustrations with mda and rm-odp*, Laboratory of Systemic Modeling, Swiss Federal Institute of Technology – Lausanne, <http://lcawww.epfl.ch/Publications/Naumenko/NaumenkoW03.pdf>

[ATL] The Eclipse Foundation, ATL Subproject, www.eclipse.org/gmt/atl/

[ATL Desc.] Model transformation with ATL, ATLAS group (INRIA & LINA), University of Nantes, France, <http://www.sciences.univ-nantes.fr/lina/atl/>

[EMF] The Eclipse Foundation, Eclipse Modeling Framework (EMF), www.eclipse.org/emf/

- [5] Jean Bezin, *Towards a Precise Definition of the OMG/MDA Framework*, <http://www.sciences.univ-nantes.fr/lina/atl/www/papers/ASE01.OG.JB.pdf>
- [6] The official MDA Guide Version 1.0.1, <http://www.omg.org/docs/omg/03-06-01.pdf>
- [7] Jean Vanderdonckt, *A MDA-Compliant Environment for Developing User Interfaces of Information Systems*
- [MOF] OMG's Meta-Object Facility: <http://www.omg.org/mof/>
- [QVT] OMG, *MOF QVT Final Adopted Specification, 2005*, <http://www.omg.org/docs/ptc/05-11-01.pdf>
- [XMI] OMG, *The XMI recommendation, 2002*, <http://www.omg.org/cgi-bin/doc?formal/2003-05-02>
- [9] OMG, *Model-Driven Architecture: Vision, Standards And Emerging Technologies*, http://www.omg.org/mda/mda_files/Model-Driven_Architecture.pdf
- [Schaefer] 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](http://www.usixml.org/index.php5?mod=download&file=Schaefer-HCIInt2007.pdf) (Beijing, 22-27 July 2007), Part I, Lecture Notes in Computer Science, Vol. 4550, Springer-Verlag, Berlin, 2007, pp. 1178-1187. <http://www.usixml.org/index.php5?mod=download&file=Schaefer-HCIInt2007.pdf>
- [10] Jean Bezin, Fabian Buttner, Martin Gogolla, Frederic Jouault, Ivan Kurtev, Arne Lindow, *Model Transformations? Transformation Models!*, University of Nantes, Computer Science Department & INRIA(A), University of Bremen, Computer Science Department & TZI(B)
- [11] Shane Sendall and Wojtek Kozaczynski, *Model Transformation – the Heart and Soul of Model-Driven Software Development*, Swiss Federal Institute of Technology in Lausanne
- [12] Aditya Agrawal, *Metamodel Based Model Transformation Language*, Institute for Software Integrated Systems (ISIS) Vanderbilt University Nashville, TN – 37235, 2003
- [13] Anonymous author, *Model Driven Architecture* on Wikipedia, http://en.wikipedia.org/wiki/Model-driven_architecture
- [14] John D. Poole, *Model-Driven Architecture: Vision, Standards And Emerging Technologies*, Position Paper Submitted to ECOOP 2001 Workshop on Metamodeling and Adaptive Object Models, april 2001
- [15] M. Bohlen, *QVT and multi metamodel transformations in MDA*, 2006, <http://galaxy.andromda.org/jira/secure/attachment/10780/QVT+article+mbohlen+2006.pdf>

- [16] Johanna Ambrosio, *Tools for the code generation*, 2003, <http://www.adtmag.com/article.aspx?id=7850&page=>
- [17] Mikko Kontio, *Architectural manifesto: Choosing MDA tools, Three categories for evaluation*, sep. 2005, <http://www-128.ibm.com/developerworks/wireless/library/wi-arch18.html>
- [18] Mike Rosen, *Which MDA Tools are Right for You?*, M2VP Inc., 2003, http://www.omg.org/news/meetings/workshops/UML_2003_Manual/03-1_Rosen.pdf
- [mdaTools] Anonymous author, *Etat de l'art des outils MDA*, <http://www2.lifl.fr/~bonde/exploration.html>
- [19] João Paulo Almeida, Luís Ferreira Pires and Marten van Sinderen, *Costs and Benefits of Multiple Levels of Models in MDA Development*, Centre for Telematics and Information Technology, University of Twente PO Box 217, 7500 AE Enschede, The Netherlands, <http://www.cs.kent.ac.uk/projects/kmf/mdaworkshop/submissions/Almeida.pdf>
- [20] Jean Bézivin, *From Object Composition to Model Transformation with the MDA*, University of Nantes, <http://www.sciences.univ-nantes.fr/info/lrsg/Recherche/mda/TOOLS.USA.pdf>
- [21] Desmond DSouza, *Model-Driven Architecture and Integration*, Opportunities and Challenges, Version 1.1, 2002, <ftp://ftp.omg.org/pub/docs/ab/01-03-02.pdf>

Appendix A

Here we give the detailed description of the UsiXML interface description language.

2.3.1.1 Task Model

The *Task Model* describes the interactive tasks as viewed by the end user interacting with the system. The task model is expressed here according to our extended version of ConcurTaskTree notation [Pate97]. The Task Model is composed of *tasks* and *task relationships*. *Tasks* are, notably, described with attributes like *name* and *type*. The *name* of the task is generally expressed as a combination of a verb and a substantive (e.g., consult patient file). The *type* refers to four basic types of tasks: user's, interactive, system and abstract task. For leaf task we consider two attributes (i.e., *userAction* and *taskItem*) that enable a refined expression of the nature of the task. This expression is based on the taxonomy introduced by [Cons03] that allows to qualify a UI in terms of abstract actions it supports. The *userAction* is represented by a verb that indicates a user action required to perform the task and the *taskItem* which refers to a type object or subject of an action. The possible values and their associated definition are presented in Section 2.3.1.2.

Task relationships are relationships involving several occurrences of different (or the same in some cases) tasks. Task relationships are of two main types:

- *Decomposition*: enables to represent a hierarchical structure of the task tree. Decomposition relationship is implicit within the XML syntax of the language and it is represented by simple embedding of elements.
- *Temporal*: allows specifying temporal relationships between tasks. We use LOTOS operators as they have been applied to task modeling in [Pate97].

A Transformational Approach for Developing Multimodal Web User Interfaces

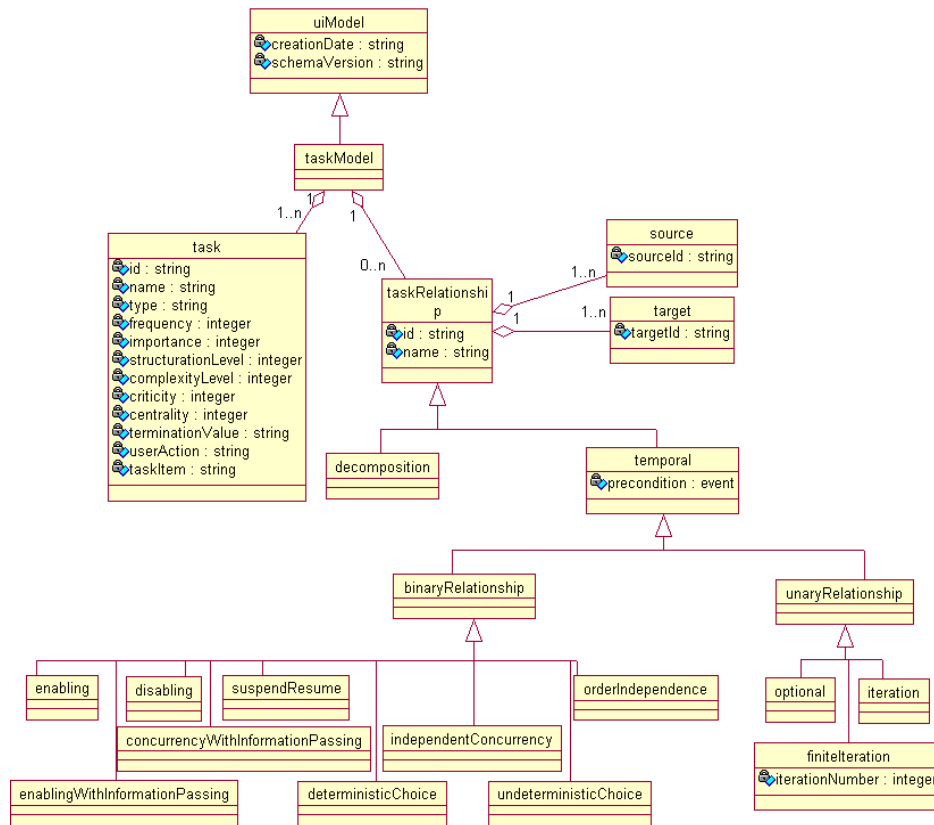


Figure 8-1. Meta-model of the Task Model

2.3.1.2 Domain Model

The *Domain Model* is a description of the classes of objects manipulated by a user while interacting with a system. It consists of one or many *domainClasses*, and potentially one or many *domainRelationships* between these classes.

A *class* describes the characteristics of a set of objects sharing a set of common properties. The concepts identified at the level of a class are the following: *attributes*, *methods*, and *objects*. An *attribute* is a particular characteristic of a class. Attributes are further described by the elements constituting the attribute class. The *attributeDataType* refers to basic data types as string, integer, real, boolean or enumerated. An *enumeratedValue* describes in extension an attribute that has the characteristic of being enumerated. The *attributeCardMin* and *attributeCardMax* describes, respectively, the lower and upper bound of the attribute cardinality (0 means that the attribute is not mandatory, 1 means that it is mandatory). A *method* is the description of a process able to change the system's state. Here, the methods are described by its signature (i.e., its name, input and output parameter(s)). An *object* is an instance of a class and is composed of attribute instances and can call methods.

A *domainRelationship* describes various types of relationships between classes. They can be classified in three types: generalization, aggregation, ad hoc. Class relationships are described with several attributes that enable to specify its role names and cardinalities.

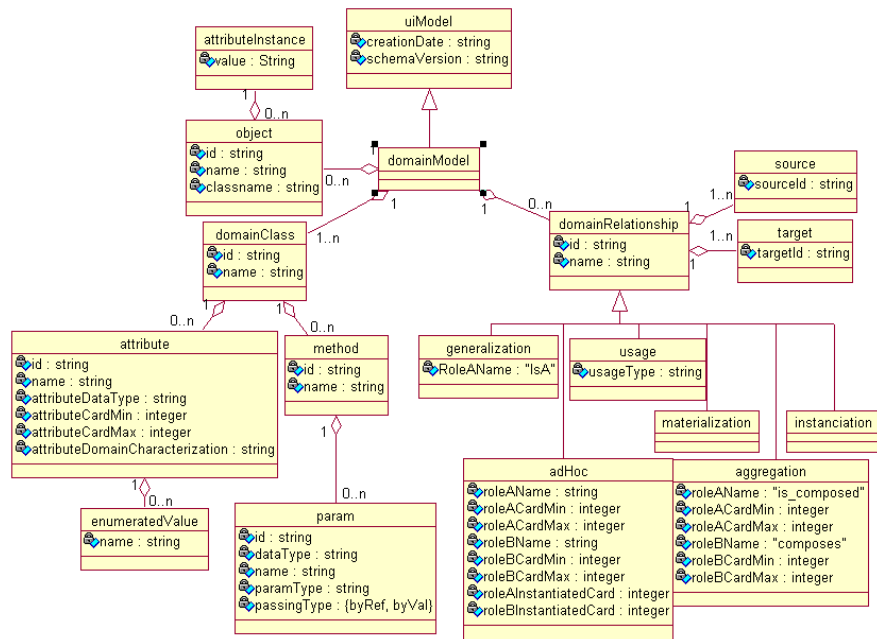


Figure 8-2. Meta-model of the Domain Model

2.3.1.3 Abstract User Interface Model

Abstract User Interface (AUI) Model is a model that represents a canonical expression of the renderings and manipulations of the domain concepts and functions in a way that is independent of any interaction modality and computing platform. As an AUI does not refer to any particular modality, we do not know yet how this abstract description will be concretized: graphical, vocal or multimodal. This is achieved in the next level.

AUI Model (Figure 8-1) is populated by *Abstract Interaction Objects (AIO)* and *Abstract User Interface Relationships* between them.

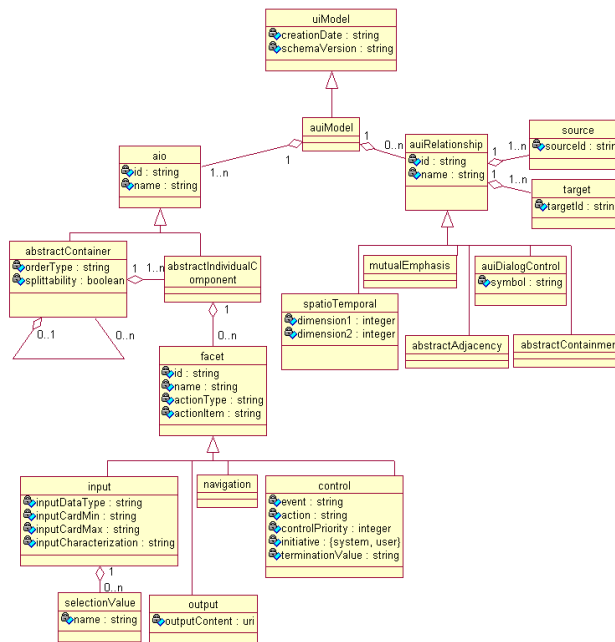


Figure 8-3. Meta-model of the AUI Model

An *AIO* is an element populating an AUI model consisting in an abstraction of widgets found in graphical toolkits (e.g., windows, buttons) and vocal toolkits (e.g., prompts, vocal menus). It can be of two types: *Abstract Individual Component (AIC)* or *Abstract Container (AC)*.

An *AIC* is any individual element populating an AC. An AIC assumes at least one basic system interaction function described as *facet* in the UI. As AICs are composed of multiple facets, we call them multi-faceted. Each facet describes a particular function an AIC may assume. We identify four main facets:

1. *Input facet*: describes the type of input that may be accepted by an AIC
2. *Output facet*: describes what data may be presented to the user by an AIC
3. *Navigation facet*: describes the possible container transition a particular AIC may enable
4. *Control facet*: describes possible methods from the Domain Model that may be triggered from a particular widget.

An AIC may assume several facets simultaneously. For instance an AIC may display an output while accepting an input from a user, trigger a container transition and a method defined in the Domain Model.

The *actionType* attribute of a facet enables the specification of the type of action an AIC allows to perform. The possible values (Table 8-1) are the same as for the *userAction* attribute of a task from the Task Model. The *view* value allows in [USIXML05] to express information by displaying it and can be reified in the Concrete level only by a graphical object. In order to keep the AUI Model independent of any modality, we replace this value by introducing in [USIX06] *convey*, a more appropriate value for *actionType* attribute, as it does not specify the employed modality.

actionType	Definition
start/go	Specifies that the AIC triggers an action
stop/exit	Specifies that the AIC puts an end to an action
select	Specifies that the AIC allows a selection action over multiple options
create	Specifies that the AIC is creating an item
delete	States that the AIC is dedicated to the deletion of items
modify	States that the AIC is dedicated to the modification of items
move	States that the AIC allows the movement of an item
duplicate	States that the AIC allows the creation of copies of an item
toggle	States that the AIC specifies the existence of two different states of an item
convey	States that the AIC expresses an information without specifying the employed modality: graphical, vocal, etc.[USIX06]

Table 8-1. Definition of possible values for the *actionType* attribute of a facet

The *actionItem* characterizes the item that is manipulated by the AIC. The possible values (Table 3-2) are identical to those of *taskItem* attribute of a task from the Task Model.

actionItem	Definition
element	Specifies that the item has a single characteristic
container	Specifies that the item is an aggregation of elements
operation	Specifies that the item is a function
collection of elements	Specifies that the item is composed of a list of elements
collection of containers	Specifies that an item is composed of a list of containers

Table 8-2. Definition of possible values for the *actionItem* attribute of a facet

By combining these two attributes a series of possible cases will appear. Table 8-3 exemplifies several possible associations.

actionType	actionItem	Example
start	operation	Search a definition of a word in an online dictionary
stop	operation	Stop searching the definition
select	element	Select the gender of a person
create	element	Input a new email address in a form
delete	collection of elements	Erase a list of phone numbers
modify	collection of containers	Modify a list of addresses
move	element	Drag and drop a predefined shape from a toolbar to the working area
duplicate	collection of elements	Copy the coordinates of a person (name, email, fax, phone number)
toggle	element	Switching on/off the connection with a network
convey	element	Express the result of a computational operation (the result can be expressed graphically by displaying it on the screen or vocally by system utterance)
convey	container	Express the starting date of a conference (the day, month and year can be displayed on the screen or can be uttered by the system)
convey	collection of elements	Express the authors list of a book (the list of authors can be displayed or can be uttered by the system)
convey	collection of containers	Express the starting and ending date of a conference (the day, month and year of the starting date and, respectively the ending date can be displayed or can be uttered by the system)

Table 8-3. Examples of combinations between actionType and actionItem attribute values

AUI Relationships are abstract relationships among AUI objects. Relationships may have multiple sources and multiple targets. There are a couple of types of relationships, between which:

- *AbstractAdjacency*: allows to specify an adjacency constraint between two AIOs
- *AbstractContainment*: allows to specify that an AC embeds one or more ACs or one or more AICs
- *AuiDialogControl*: enables the specification of a dialog control in terms of LOTOS operators between AIOs.

2.3.1.4 Concrete User Interface Model

Concrete User Interface (CUI) Model is a model that allows the specification of the presentation and behavior of a UI with elements that can be perceived by the users [Limb04b]. The CUI abstracts a Final UI in a definition that is independent of programming toolkit peculiarities.

CUI Model (Figure 8-2) concretizes the AUI for a given context of use into *Concrete Interaction Objects/Components (CIOs/Components)* and *Concrete User Interface Relationships* so as to define layout and/or interface navigation of 2D graphical widgets and/or vocal widgets.

CIOs realize an abstraction of widget sets found in popular graphical and vocal toolkits (e.g., Java AWT/Swing, HTML 4.0, Flash DRK 6, VoiceXML). A CIO is defined as an entity that users can perceive and/or manipulate (e.g., window, push button, text field, check box, vocal output, vocal input, vocal menu). Because UsiXML considers both graphical and vocal modalities, CIOs are further divided into two types: *graphicalCIOs* and *vocalCIOs*. A detailed explanation regarding the types of *graphicalCIOs*, *vocalCIOs* and the corresponding *Concrete User Interface Relationships* between them, along with their semantics and syntax is presented in the following sections.

Any CIO can have any number of *behaviors*. A *behavior* is the description of the triplet *event-action-condition* that determines the UI change. In the following we offer a brief description for each of the terms involved in the triple, but a more detailed documentation can be found in [USIX06]:

- *Event*: specifies an expression triggering one or several actions. Events are restricted to a specific event language. Graphical *eventsTypes* are described in Section 8.3.2.1. The attribute *eventContext* allows mentioning the concerned CIO, depending on the type of event. The attribute *device* is a reference to the device with which the event is triggered.
- *Action*: is a process triggered by an event performed on a CIO. An action may be a method call, a UI internal change, etc.
- *Condition*: enables to specify a pre/post-condition attached to an action. A condition is expressed as a graph patters (i.e., a rule term) that must be fulfilled in the specification before or after the application of an action. Conditions may be combined with Boolean operators to compose complex conditions.

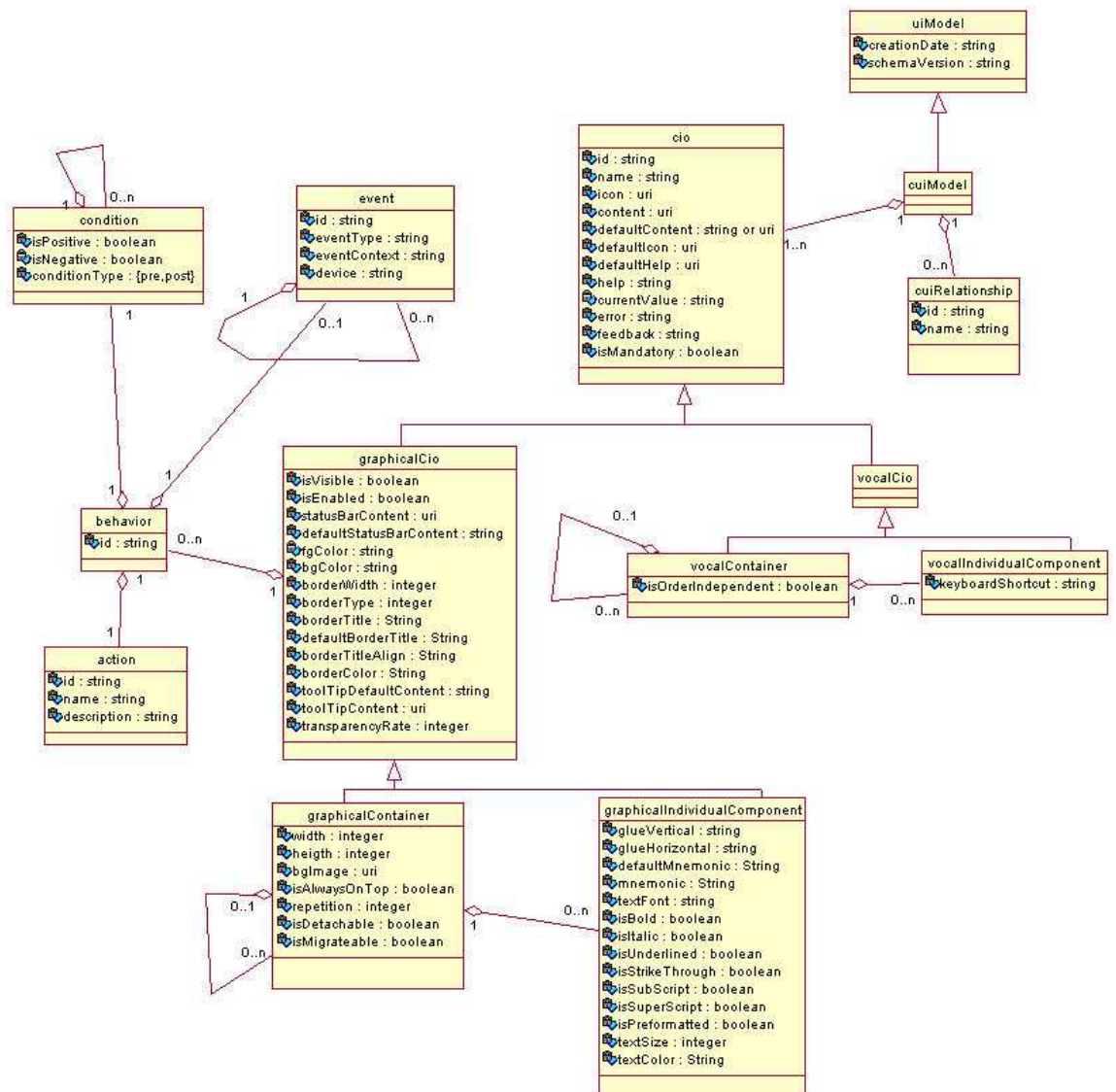


Figure 8-4. Excerpt of the CUI Meta-model

2.3.1.5 Final User Interface

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

2.3.1.6 Context Model

The *Context Model* (Figure 8-5) describes all the entities that may influence how the user's task is carrying out with the future UI. It takes into account three relevant aspects, each aspect having its own associated attributes: *user type* (e.g., experience with device and/or system, task motivation), *computing platform type* (e.g., desktop, PocketPC, PDA, GSM), and

physical environment type (e.g., lighting level, stress level, noise level). These attributes initiate transformations that are applicable depending on the current context of use.

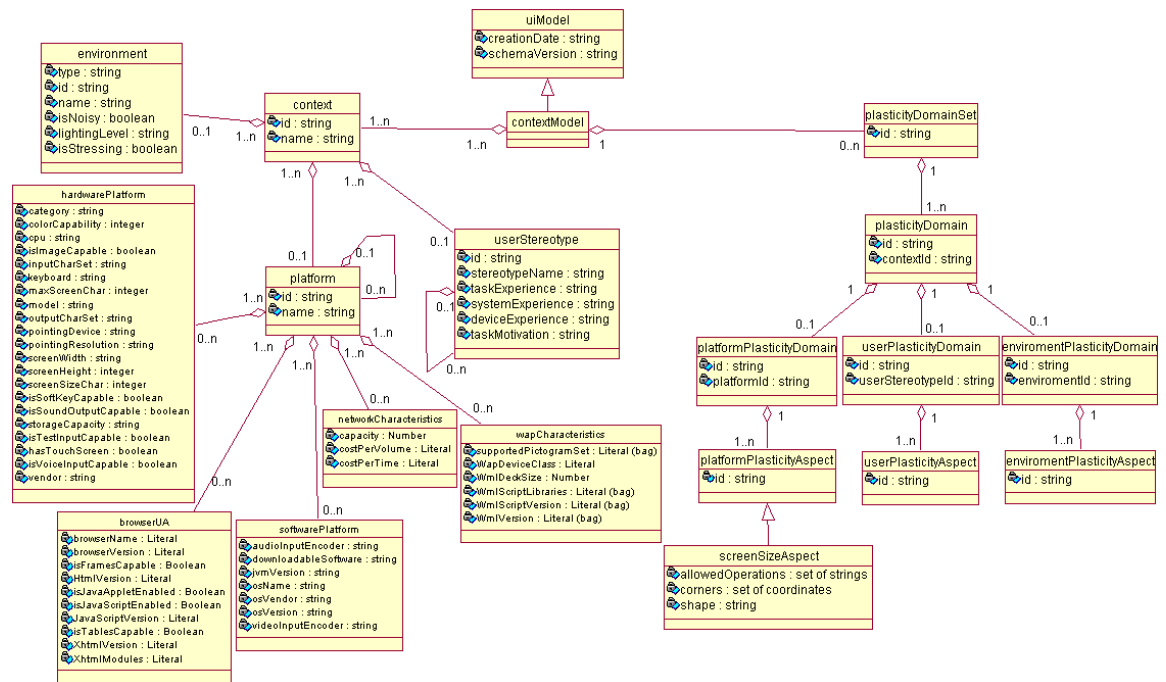


Figure 8-5. Meta-model of the Context Model

2.3.1.7 Mapping Model

The *Mapping Model* (Figure 8-6) contains a series of related mappings between models or elements of models. A mapping model serves to gather a set of pre-defined, inter-model relationships that are semantically related. It consists of one to many *interModelRelationships*, a part of them being used throughout the steps of the transformational approach:

- *Manipulates*: maps a task onto a domain concept (i.e., a class, an attribute, a method or any combination of these types).
- *Updates*: is a mapping between any UI component (at abstract or concrete level) and a domain attribute or instantiated attribute (at run time). Updates enables to specify that a UI component provides a value for the related domain concept.
- *Triggers*: indicates a connection between a method of the Domain Model and a UI individual component (either at the abstract or at the concrete level)
- *IsExecutedIn*: indicates that a task is performed through one or several ACs and AICs.
- *IsReifiedBy*: maps the elements of an AUI onto elements of a CUI. This relationship specifies how any AIO can be reified by a CIO.

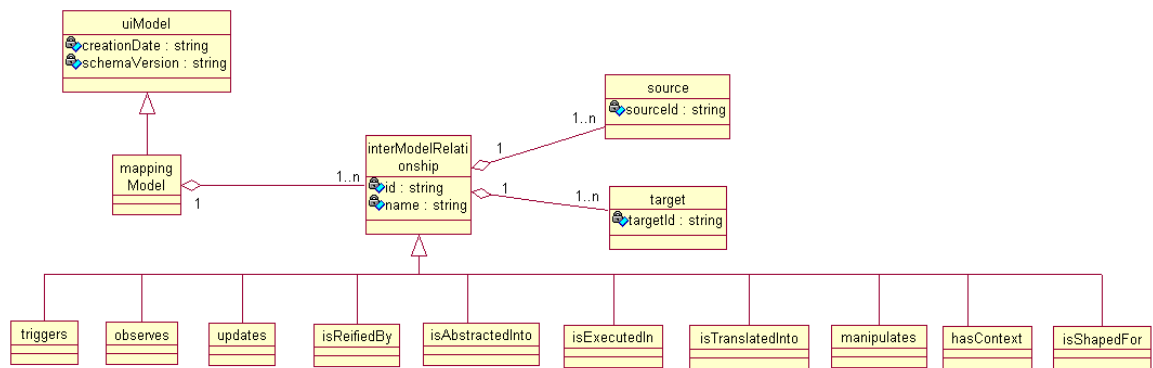


Figure 8-6. Meta-model of the Mapping Model

2.3.1.8 Transformation Model

Transformation Model (Figure 8-7) contains a set of rules enabling the transformation of one specification (at a certain level of abstraction) into another or to adapt a specification for a new context of use. A transformation rule realizes a unit transformation operation on a model. It is composed of a:

- *Lhs (Left Hand Side)*: models the pattern that will be matched in the host model
- *Rhs (Right Hand Side)*: models the part that will replace the LHS in the host model
- *NAC (Negative Application Condition)*: models the condition that have to hold false before trying to match LHS into the host model
- *AttributeCondition*: is a textual expression indicating a condition scoping on element attributes of the lhs of a transformation rule
- *RuleMapping*: defines the source and the target models of the transformation rule. For instance, a rule may establish a mapping between a *Task Model* and an *Abstract Model*. In this case, the source indicates the source model of the mapping, while the target indicates the target model.

Transformation rules are applied in order to develop UIs following a specific development path (e.g., forward engineering, reverse engineering, adaptation to context of use). A development path is composed of development steps that can imply three types of transformations depending on the development direction:

- *Reification*: consists in the derivation of the next lower model in our reference framework
- *Abstraction*: consists in the derivation of the next upper model in our reference framework
- *Translation*: is a type of model transformation adapting a set of UI models to a target context of use.

A development step is decomposed into development sub-steps. A development sub-step is realized by one (and only one) transformation system. A transformation system is composed of a set of sequentially applied transformation rules. One transformation system applies one sub-derivation unit [Limb04]. A sub-derivation unit is defined as a collection of derivation rules that realize a basic development activity. A basic development activity has been identified to

sub-goals assumed by the developer while constructing a system, for instance choosing widgets, defining navigation structure, etc

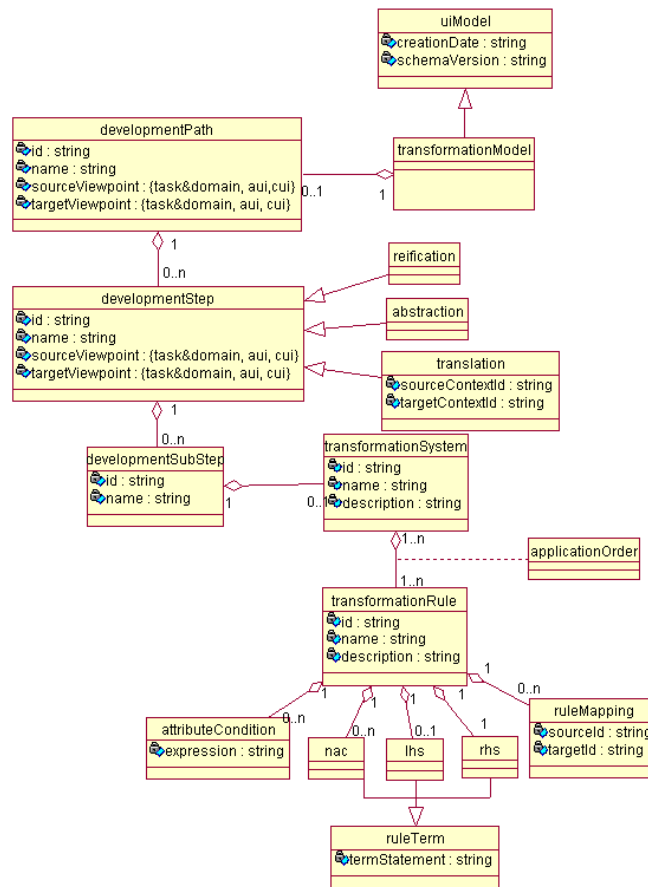


Figure 8-7. Meta-model of the Transformation Model

Appendix B

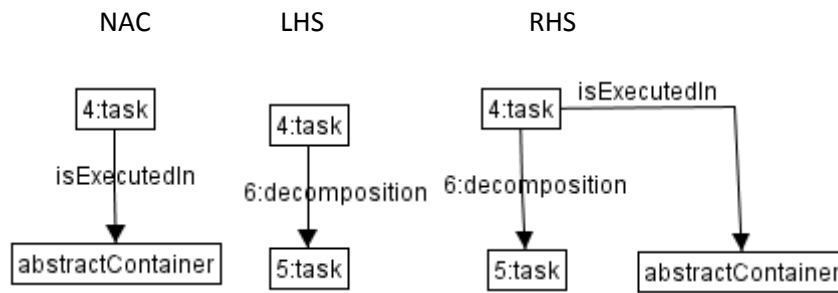


Figure 9-1. Rule 1: Create an AC for task that has task children

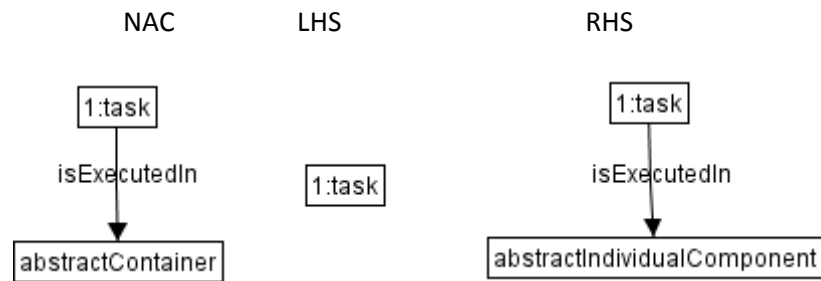


Figure 9-2. Rule 2: Create an AIC for leaf tasks

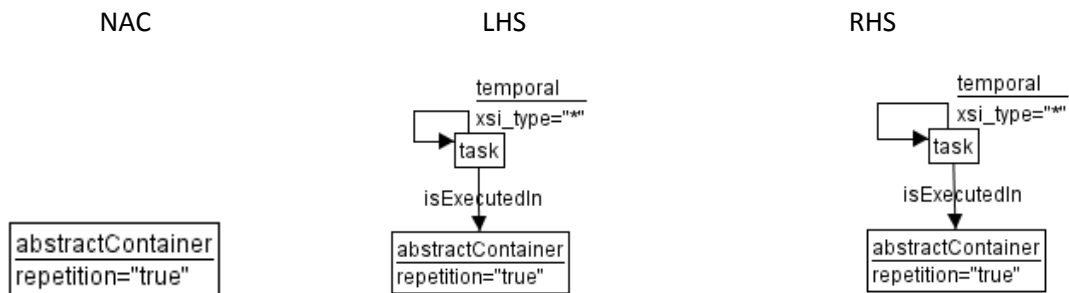


Figure 9-3. Rule 3: Iterative tasks are mapped onto repetitive AC

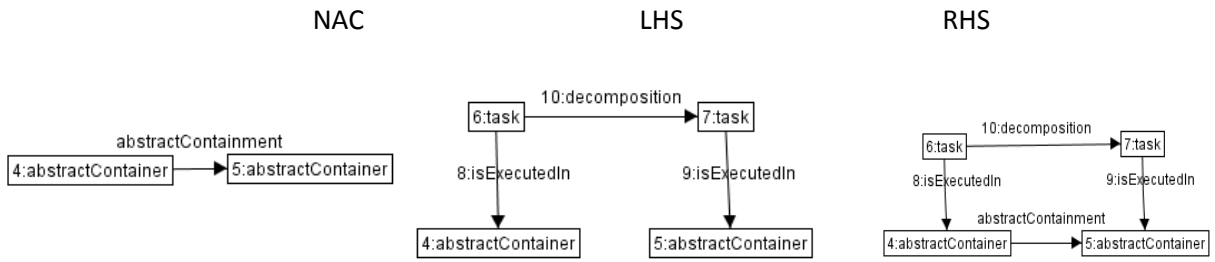


Figure 9-4. Rule 4: Reconstruct containment relationship between ACs

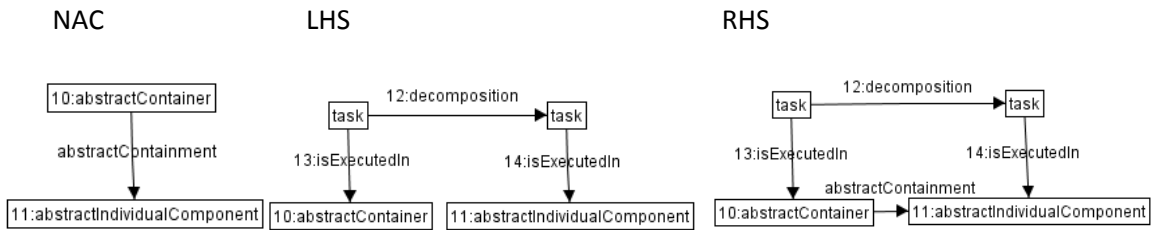


Figure 9-5. Rule 5: Reconstruct containment relationship between ACs and AICs

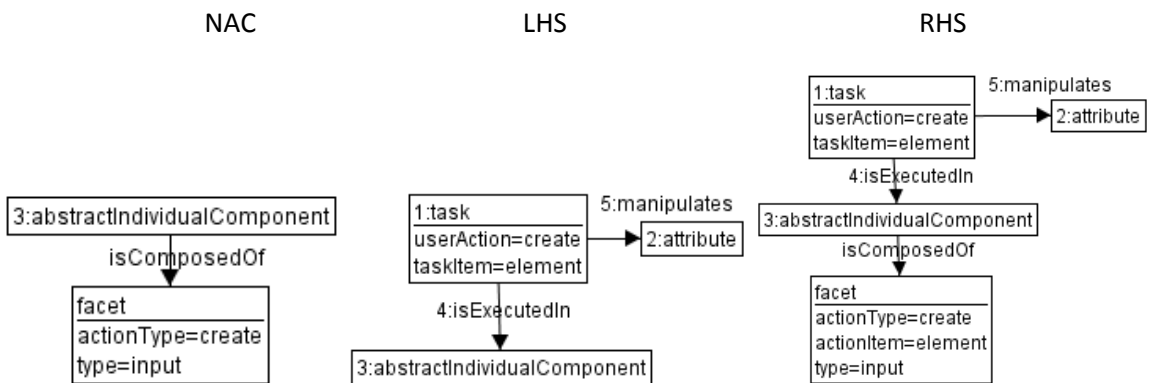


Figure 9-6. Rule 6: Create an input facet for AIC executed in tasks of type create

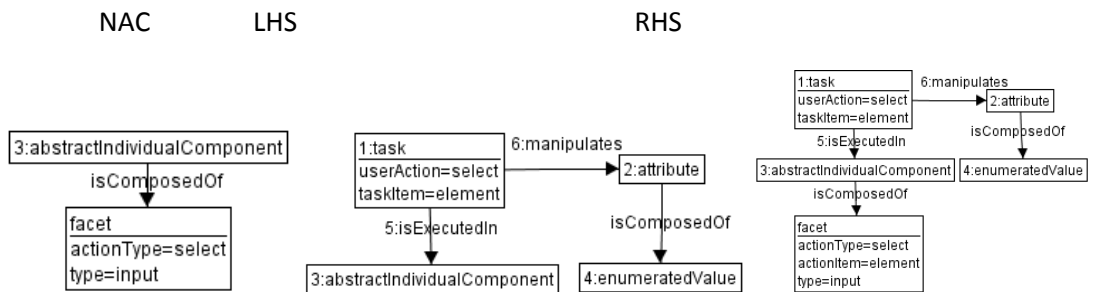


Figure 9-7. Rule 7: Create an input facet of type select element when an enumerated value attribute is encountered

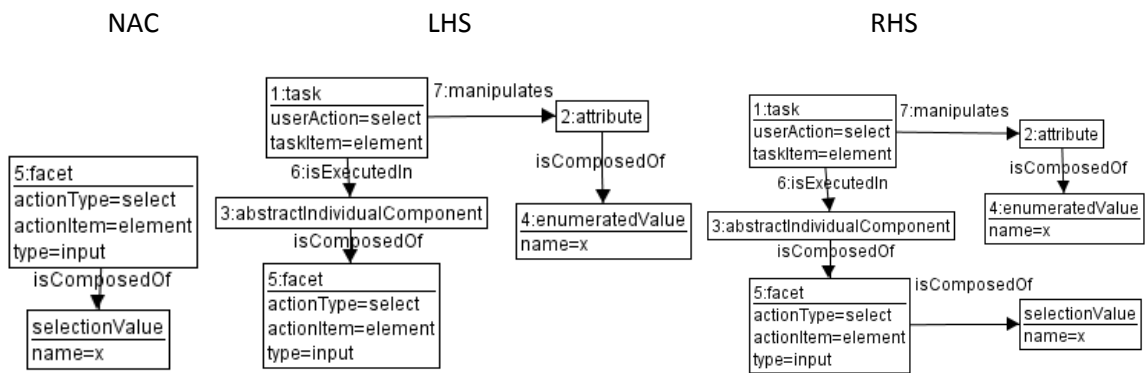


Figure 9-8. Rule 8: Create selection values for facets of type select for each enumerated value of an attribute

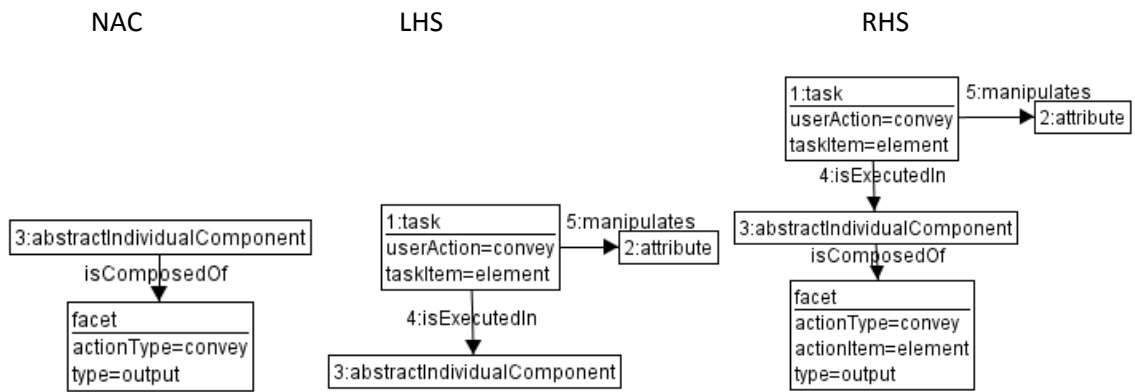


Figure 9-9. Rule 9: Create an output facet that conveys an element

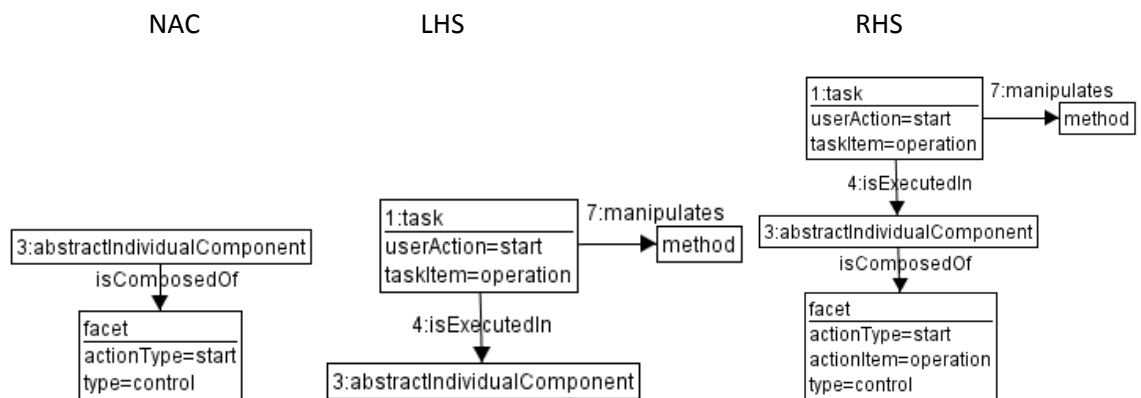


Figure 9-10. Rule 10: Create a control facet of type start operation when a method is manipulated by a task

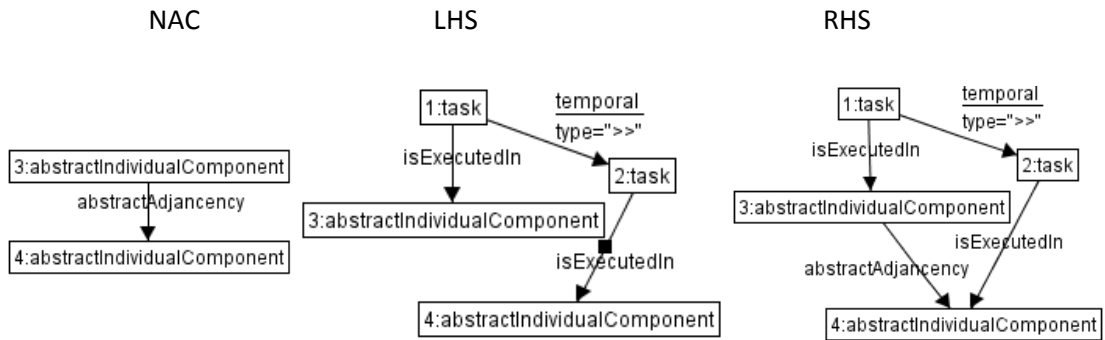


Figure 9-11. Rule 11: Creating abstract adjacency for <AIC, AIC> couple

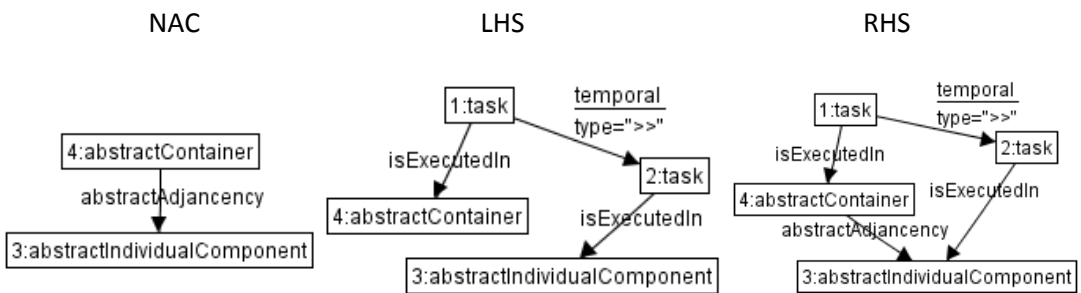


Figure 9-12. Rule 12: Creating abstract adjacency for <AC, AIC> couple

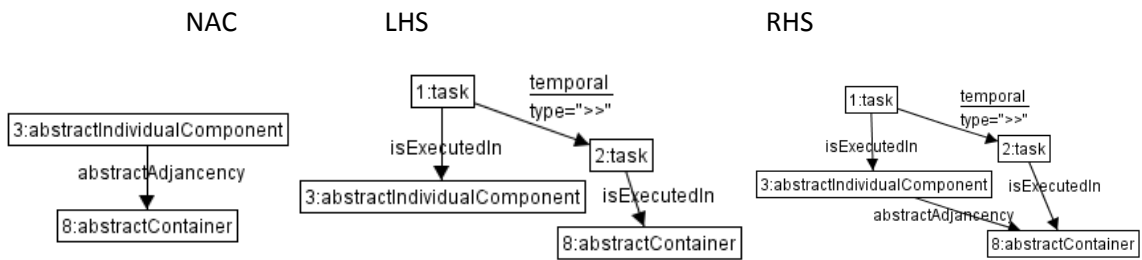


Figure 9-13. Rule 13: Creating abstract adjacency for <AIC, AC> couple

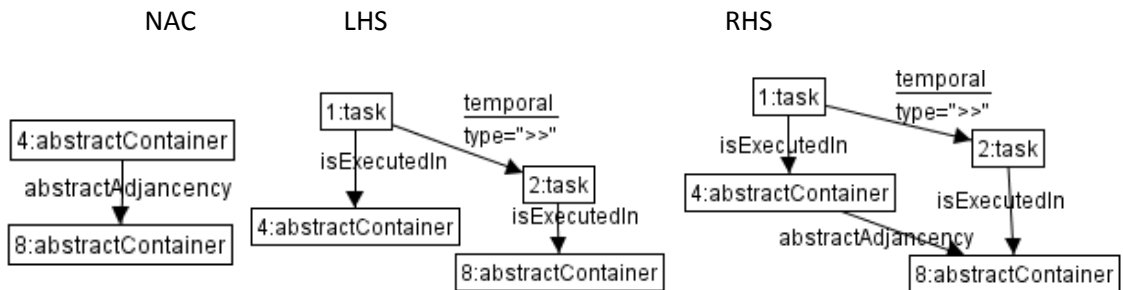


Figure 9-14. Rule 14: Creating abstract adjacency for <AC, AC> couple

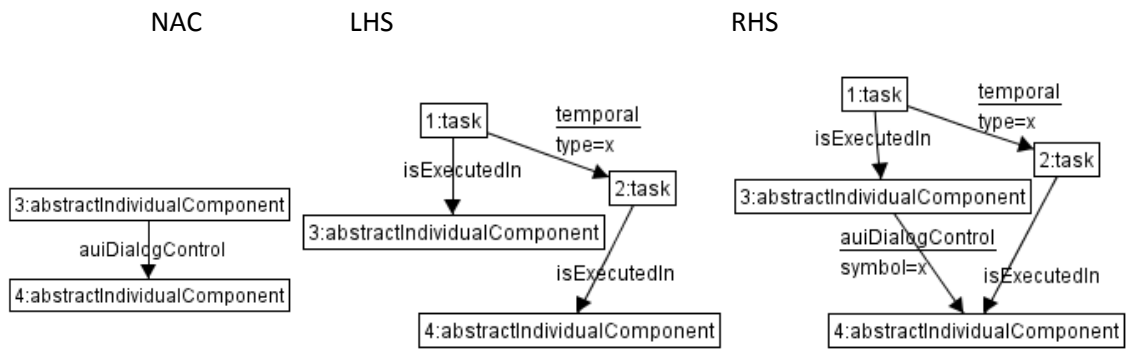


Figure 9-15. Rule 15: Deriving Abstract Dialog Control for <AIC, AIC> couple

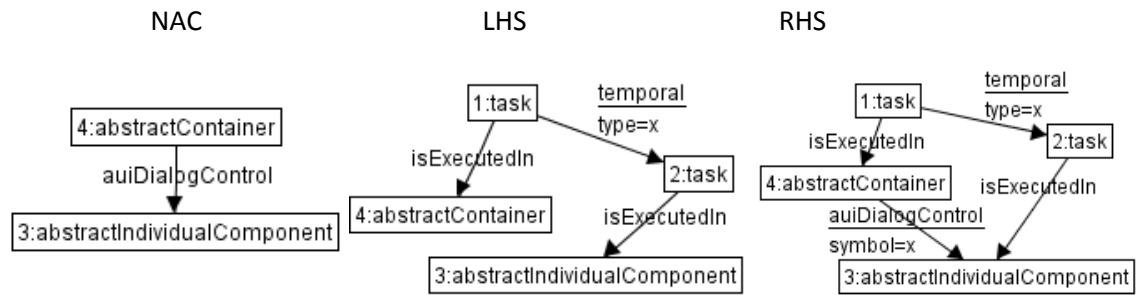


Figure 9-16. Rule 16: Deriving Abstract Dialog Control for <AC, AIC> couple

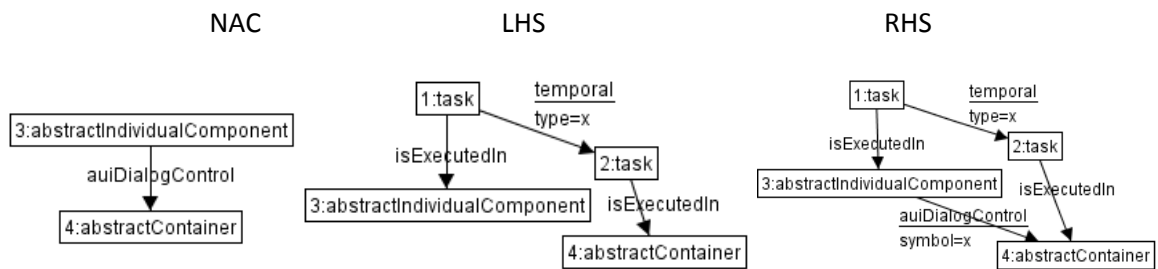


Figure 9-17. Rule 17: Deriving Abstract Dialog Control for <AIC, AC> couple

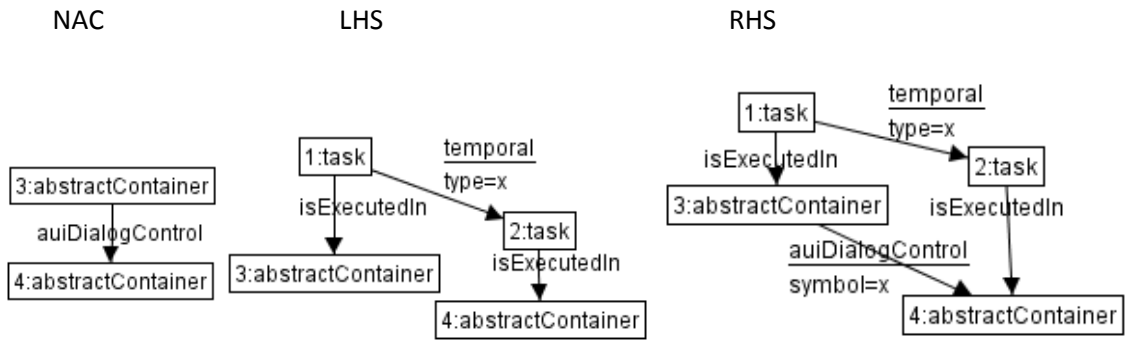


Figure 9-18. Rule 18: Deriving Abstract Dialog Control for <AC, AC> couple

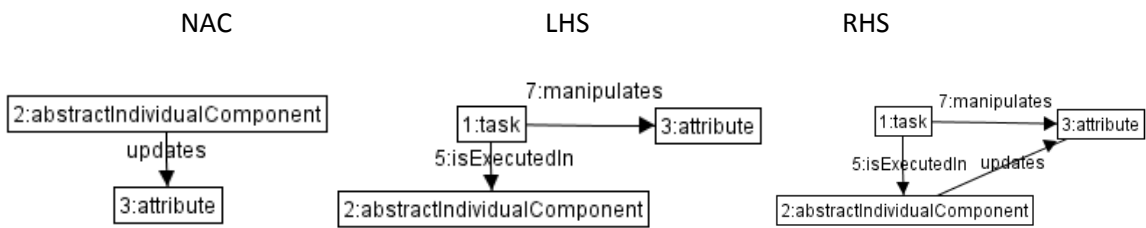


Figure 9-19. Rule 19: Deriving updates relationships for an AIC

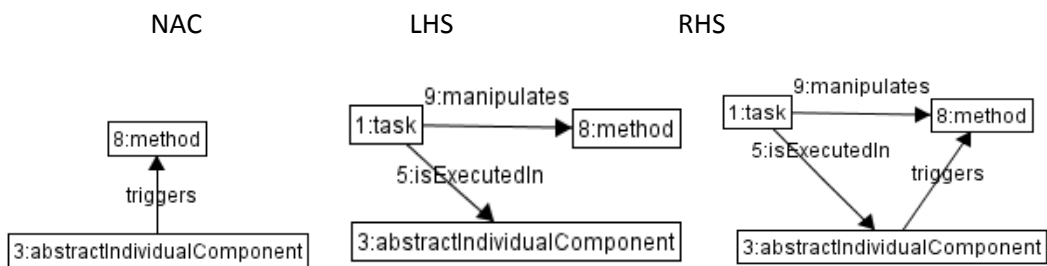


Figure 9-20. Rule 20: Deriving trigger relationships for AICs

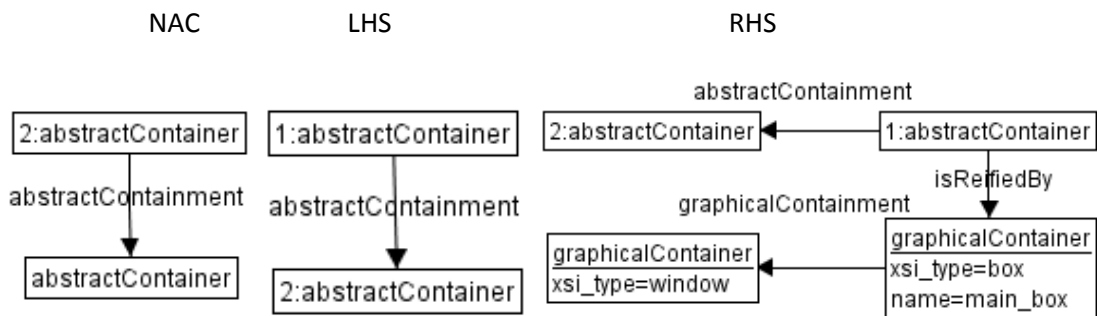


Figure 9-21. Rule 21: Creation of windows derived from abstract containment relationship

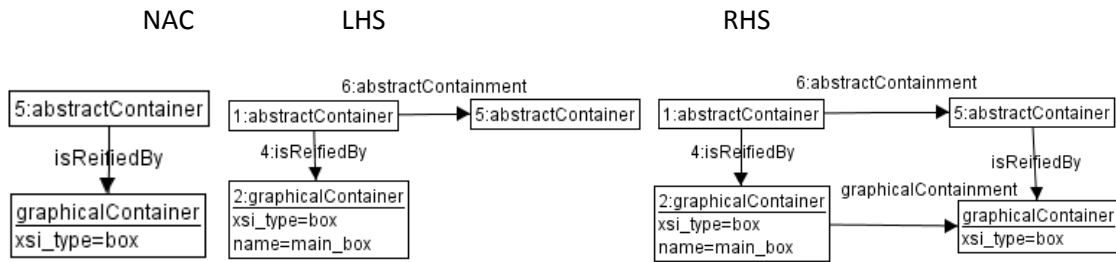


Figure 9-22. Rule 22: Generation of graphical containers of type box

Sub-step 3.2: Selection of CICs

The rule illustrated in Figure 8-23 generates a GC of type **box** that will embed two GICs: an **outputText** and an **inputText** representing respectively the label and the associated text field.

Figure 5-24 describes the rule applied in order to create a GC of type **box** that will embed a **group of radio buttons** when an input facet of type select element is encountered. A GIC of type **outputText** representing the label associated to this group is also created. The **defaultContent** of the GIC is the same as the name of the AIC. The radio buttons associated to this group are created by executing the rule described in Figure 5-25. For each selection value of a facet of type select, a radio button, that has the same content as the name of the selection value, is created

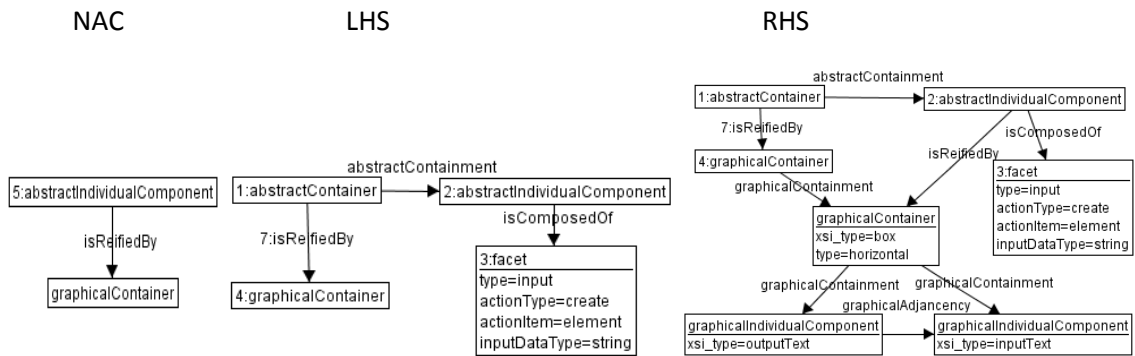


Figure 9-23. Rule 23: Generation of an outputText and an inputText for AIC with create input facet

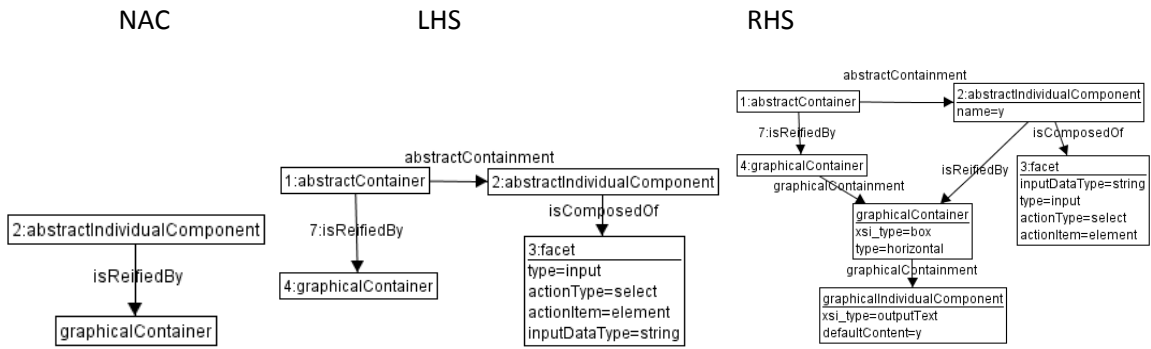


Figure 9-24. Rule 24: Generation of a graphical container of type box that will contain a group of radio buttons

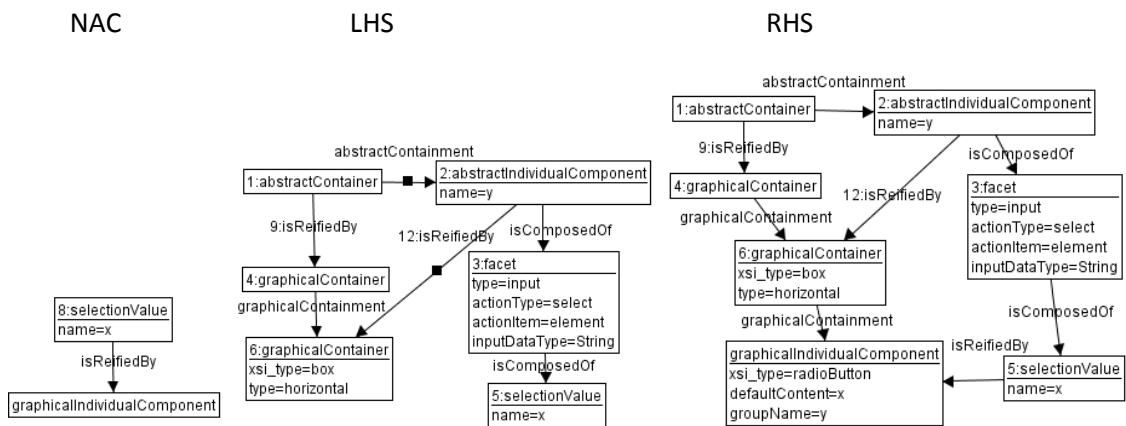


Figure 9-25. Rule 25: Generation of radioButtons for each selection value of a facet of type select

Figure 8-26 presents the rule applied to generate a GC of type **outputText**, each time an output facet of type create is encountered.

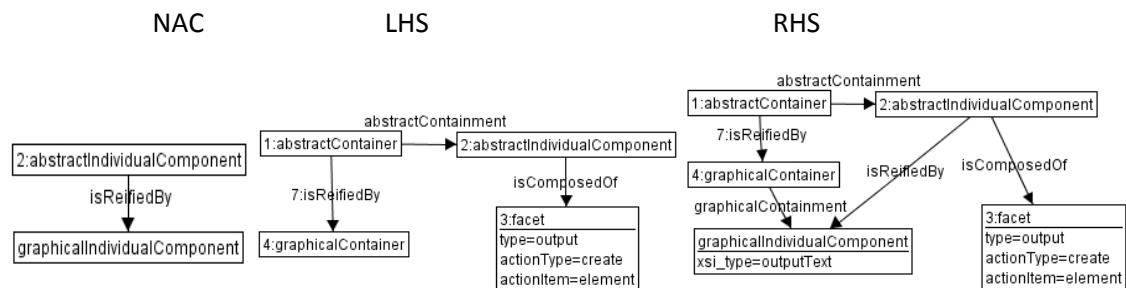


Figure 9-26. Rule 26: Generation of an outputText for an output facet with create action type

The GIC of type **button** is created when a control facet of type start operation is encountered.

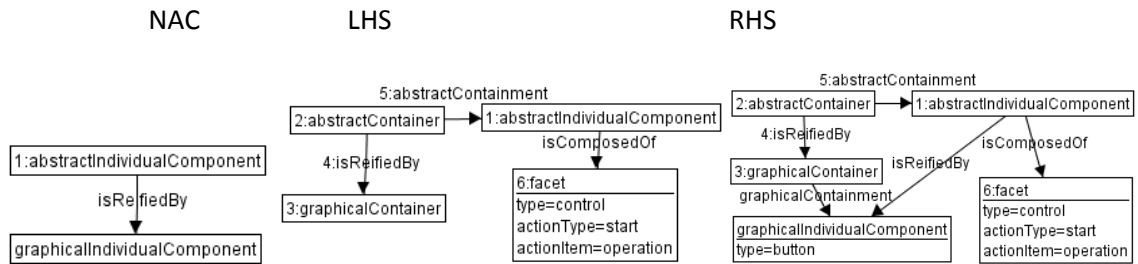


Figure 9-27. Rule 27: Generation of a control button

Sub-step 3.3: Arrangement of CICs

For each couple of adjacent AIOs that are reified into graphicalCICs, we define a graphicalAdjacency relationship between these graphicalCICs. As AIOs can be of two types (i.e., ACs or AICs), there are four possible combination to take into account. For each combination a specific rule is considered (Figures 8-28, 8-29, 8-30 and 8-31).

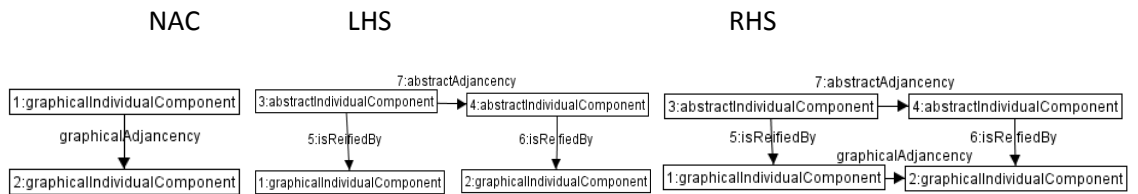


Figure 9-28. Rule 28: Generation of Graphical Adjacency relationships for <GIC, GIC> couples

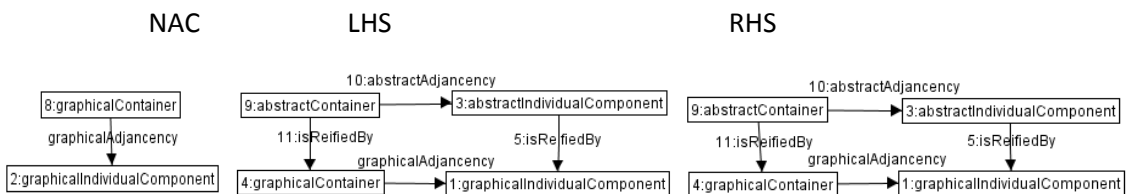


Figure 9-29. Rule 29: Generation of Graphical Adjacency relationships for <GC, GIC> couples

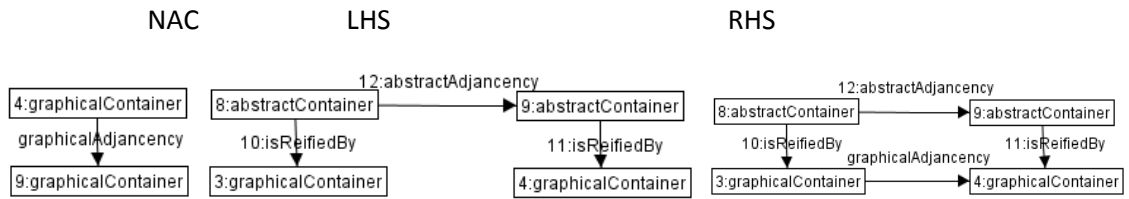


Figure 9-30. Rule 30: Generation of Graphical Adjacency relationships for <GC, GC> couples

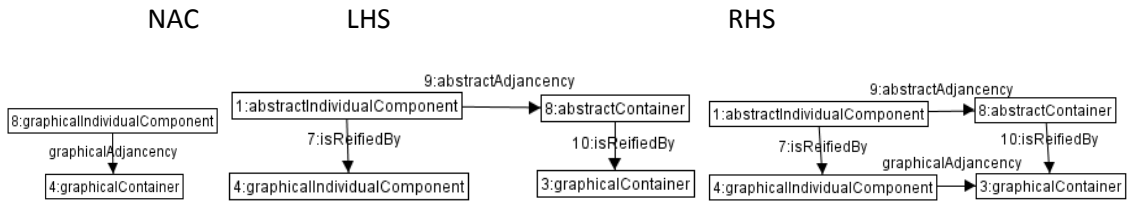


Figure 9-31. Rule 31: Generation of Graphical Adjacency relationships for <GIC, GC> couples

Sub-step 3.4: Navigation definition

The rules that ensure the navigation definition are not exemplified for this case study as all the components of the virtual polling system are presented into the same window.

Sub-step 3.5: Concrete Dialog Control Definition

For each couple of AIOs with a dialog control relationship, a transposition of this relationship to the graphicalCIOs that reify them is realized. As AIOs are of two types (i.e., ACs and AICs), four rules describing the four possible combinations are considered (Figures 8-32, 8-33, 8-34 and 8-35).

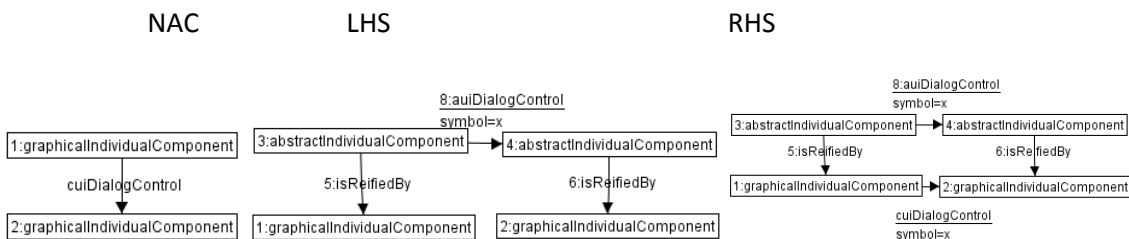


Figure 9-32. Rule 32: Generation of Concrete Dialog Control relationships for <GIC, GIC> couples

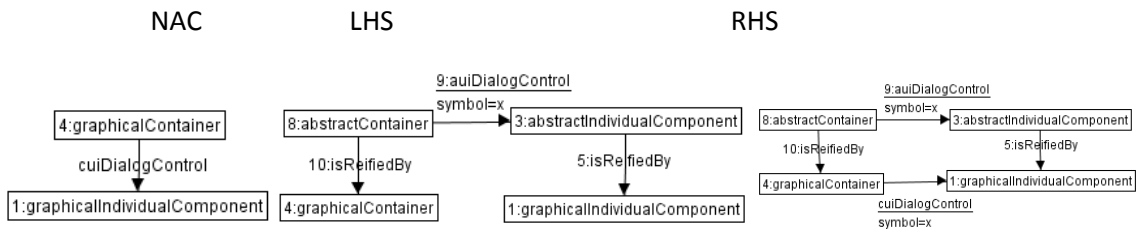


Figure 9-33. Rule 33: Generation of Concrete Dialog Control relationships for <GC, GIC> couples

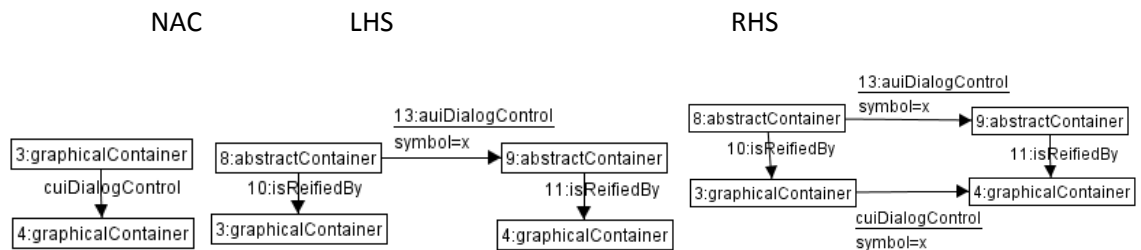


Figure 9-34. Rule 34: Generation of Concrete Dialog Control relationships for <GC, GC> couples

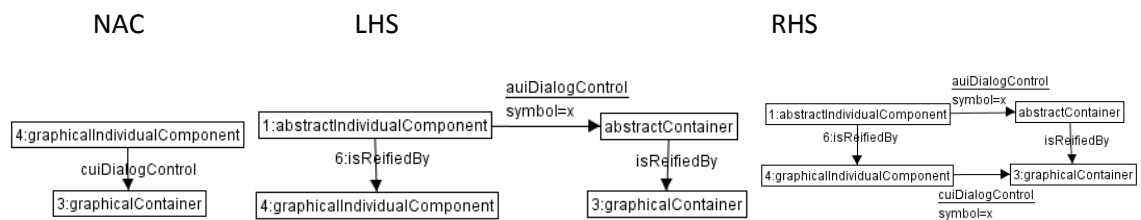


Figure 9-35. Rule 35: Generation of Concrete Dialog Control relationships for <GIC, GC> couples

Sub-step 3.6: Derivation of CUI to Domain Relationship

Figure 8-36 illustrates the rule used to map GICs with the corresponding attribute of an object from the Domain Model. The **updates** relationship is transposed from the AIC that is reified by the GIC.

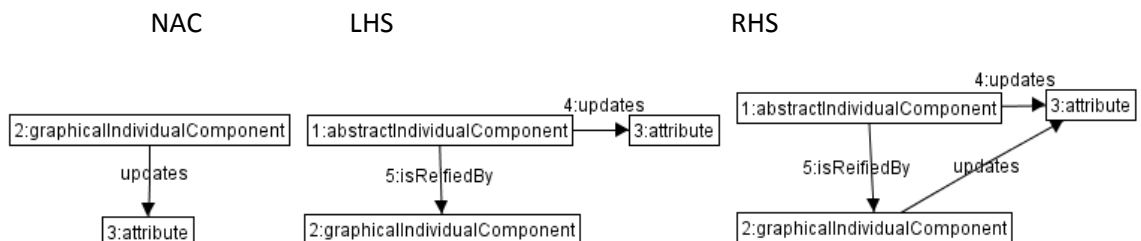


Figure 9-36. Rule 36: Transposition of update relationship

Figure 8-37 illustrates the rule used to map GICs with the corresponding method of an object from the Domain Model. The **triggers** relationship is transposed from the AIC that is reified by the GIC.

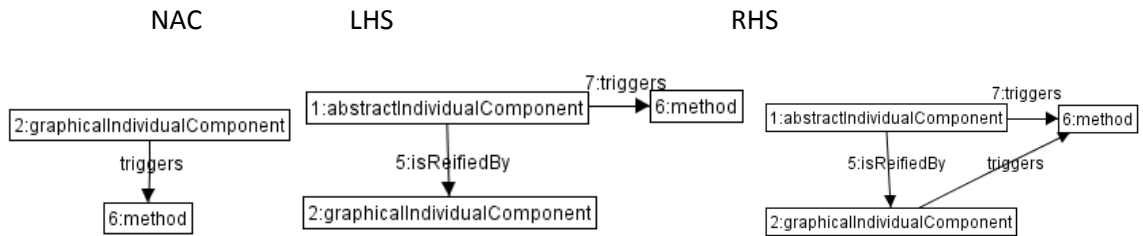


Figure 9-37. Rule 37: Transposition of triggers relationship