INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

UMLAUT: an Extendible UML Transformation Framework

Wai Ming Ho , Jean-Marc Jézéquel , Alain Le Guennec , François Pennaneac'h

N°3775

Octobre, 1999

THÈME 1

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--3775--FR+ENG

UMLAUT: an Extendible UML Transformation Framework*

Wai Ming Ho , Jean-Marc Jézéquel , Alain Le Guennec , François Pennaneac'h †

Thème 1 — Réseaux et systèmes
Projet PAMPA

Rapport de recherche n˚3775 — Octobre, 1999 — 16 pages

Abstract: Many UML CASE tools allow software engineers to draw diagrams and generate code skeletons from them. But often advanced users want to do more with their UML models, e.g., apply specific design patterns, generate code for embedded systems, simulate the functional and non-functional behavior of the system, run validation tools on the model, etc. which are very difficult to do with the scripting facilities offered in most UML case tools. In this paper, we describe UMLAUT, a freely available UML Transformation Framework allowing complex manipulations to be applied to a UML model. These manipulations are expressed as algebraic compositions of reified elementary transformations. They are thus open to extensions through inheritance and aggregation. To illustrate the interest of our approach, we show how the model of an UML distributed application can be automatically transformed into a labeled transition system validated using advanced protocol validation technology.

Key-words: Object-oriented Frameworks, UML, Transformation, Validation

(Résumé : tsvp)

* This work has been partially funded by CNET under the METAFOR project, contract *81B070/1*
† {waimingh, jezequel, aleguenn, pennanea}@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

UMLAUT: un Framework Extensible pour la Transformation de Modèle UML

Résumé : De nombreux Ateliers de Génie Logicile (AGL) pour UML permettent aux ingénieurs logiciels de tracer des diagrammes et de générer des squelettes de code à partir de ceux-ci. Mias souvent les utilisateurs avertis voudraient faire plus de chose avec leurs modèles UML, comme par exemple appliquer des design patterns spécifiques, générer du code pour des systèmes embarqués, simuler des aspects fonctionnels ou non fonctionnels du système, ou encore passer des outils de validation sur le modèle; activités difficiles à mener en utilisant les facilités de scriptage offertes par la plupart des AGL. Dans ce rapport, nous décrivons UMLAUT, un framework de transformation de modèles permettant d'appliquer des manipulations complexes à un modèle UML. Ces manipulations sont exprimés comme des compositions algébriques de transformations élémentaires réifiées. Elle sont donc extensibles au travers des mécanismes classiquess d'héritage et d'aggrégation. Pour illustrer l'intérêt de notre approche, nous montrons comment le modèle UML d'une application répartie peut être automatiquement transformé en un système de transitions étiquetées afin d'être validé par des outils avancés de validation de protocoles.

Mots-clé : Frameworks Orienté Objet, UML, Transformation, Validation

1 The need for formal manipulation of UML models

1.1 UML: the silver-bullet of modeling notations ?

Since it was standardized by the OMG in 1997, UML has been on its way to become also a *de-facto* standard, as support for its diagrammatic notations in object-oriented modeling tools keeps growing. Its use will range from basic applications for personal computers to large and complex software. This spectrum of UML-modelizable systems will grow up with the next projected releases of the notation. For example, future extensions to the UML will cover real-time, scheduling and performance[11], or enterprise distributed object computing (EDOC[10]). It is likely that such additions to the UML will push designers to use it for even larger, and more critical software.

1.2 Complex software needs validation and test

Unfortunately, extending the notation will not be sufficient to improve the quality of such large, distributed systems. Indeed, distributed systems raise their own issues, due to the complexity of their communication mechanisms: in the case of asynchronous communications, messages may be never delivered, race conditions or deadlocks may happen...

The reliability of these intrinsically concurrent systems can be enhanced with the use of formal techniques, such as model-checking, simulation or test generation. This is particularly true for telecommunication systems, a context that has been widely explored for several years and gave birth to standardized Formal Description Techniques (FDT) and associated tools. There is obviously a lack in the UML for similar concepts, and the integration of already existing validation techniques into the UML is not an easy process. This is mainly due to the facts that FDT don't fit well into the object-oriented concepts of the UML, have a steep learning curve and impose restrictions to the model (for example, the semantics of communication in FDT is often restricted).

This is why we advocate for the use of UML as a formal language. Of course, we must keep in mind that the UML is not as formalized as FDT, and first its semantics has to be enforced to ensure models can be validated and tested. But UML has some strong advantages :

- its notation is very expressive,
- it is an OMG adopted standard,
- it has some support for distributed systems (asynchronous calls, deployment diagrams...),
- some parts of the UML have their semantics defined.

The validation of software should not be seen as a kind of "post-phase" in the development process, but rather as a continuous activity that has its roots in the early specification phases and builds gradually, following the refinement process. This is another point in favor of a formalization of UML: the validation is achieved partly through traceability between the refined models. Such a traceability requires that a refined model and the model it derives from can be compared: formalization is the way to prove two models reflect the same specification. The next step is the generalization of such traceability links in order to build libraries of generic model transformations that can be proven equivalent for a refinement relationship.

1.3 UML in tools: what's behind the notation ?

The lack of formalization in the UML implies weaknesses in current tools, which most of the time limit themselves to powerful graphical editors with many bells and whistles, but behave poorly in the process of automating some painful and error-prone tasks, let alone validation. Code generators are good examples: the static part of the notation is often well-understood, and most tools allow for the generation of class skeletons from class diagrams. Then it's often up to the programmer (we can't speak here of a designer !) to manually fill in the gaps with hand-written code to get a complete and sometimes running program. We think the role of tools should not be limited to help the designer in the implementation phase of the development process, but to better assist him during the whole software development cycle.

Some CASE tools editors have recognized the needs for more automation capabilities and provide some scripting capabilities in their software. This allows for basic operations on models such as adding a method to many classes. But it still remains difficult to express complex operations such as the application of a design-pattern to a model.

The choice of another scripting language would not help that much; it is not only a language problem, but also a consistency problem: even in the case of an elementary operation, there is no means to check for the validity of such a transformation. Most of the time, tools allow any operation to be performed on a model, provided the result is syntactically correct. No semantics checks can be performed, no traceability from the original model is realized.

In this paper, we describe UMLAUT, a freely available UML transformation framework allowing complex manipulations to be applied to a UML model (Section 2). These manipulations are expressed as algebraic compositions of reified elementary transformations (Section 3). They are thus open to extensions through inheritance and aggregation. To illustrate the interest of our approach, we show (Section 4) how the model of an UML distributed application can be automatically transformed into a labeled transition system validated using advanced protocol validation technology. Then we conclude on the perspectives open by this approach (Section 5).

2 UMLAUT : An extendible transformation framework

UMLAUT is a tool dedicated to the manipulation of UML models. It has the ability to import and export model descriptions in various formats. The fact that it's available for free and some of the known formats are well-documented and/or standardized (such as CDIF, and XMI in the future) ensures that our tool is open and ready for being integrated as a background processor in other popular modelers. It can also work as a standalone application driven by a portable GUI built with Java and the Swing libraries.

2.1 General architecture

UMLAUT is a generic framework composed of a core engine which communicates with its surroundings via *hot-spots* (i.e.. interfaces), where functional modules can be plugged in order to specialize the behavior to meet specific requirements (see figure 1). Some ready-to-use plug-ins already exist and are provided with the tool: they cover various topics such as code-generation aspects (Eiffel or Java), communications via interchange format (CDIF or MDL), or transformations of models dedicated to the validation of distributed reactive systems.

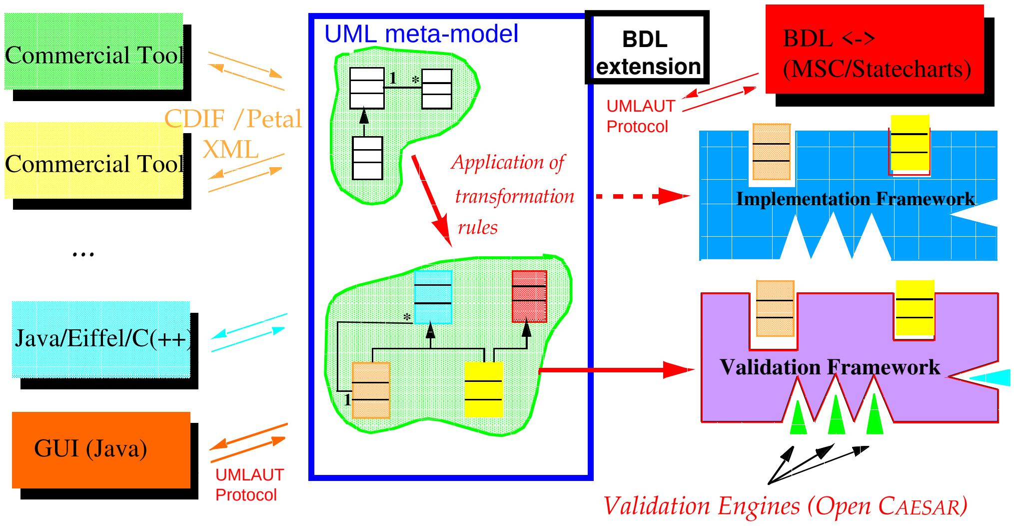

Figure 1: UMLAUT Architecture

2.2 Core engine

Basically, the UMLAUT Core Engine is a set of collaborative classes that implements the UML meta-model, as described in[9]. This meta-model is defined with a set of UML class diagrams, which contain classes and associations between these classes.

It is easy to deduce a simple code generator for such a static model: the UML classes immediately map into a set of Eiffel classes. The associations map into attributes which contain an object or a collection of objects, depending on the multiplicity of the association ends. The following example highlights this process:

- an instance of a class `UML_PACKAGE` will be created to represent a Package in a model.

- the association `ownedElement` from `Package` to `ModelElement` will expand into an attribute `ownedElement` in class `UML_PACKAGE` of type `COLLECTION[UML_MODEL_ELEMENT]`, and conversely an attribute `package` of type `UML_PACKAGE` in class `UML_MODEL_ELEMENT`.

Of course, some different implementations are possible, such as one that would reify associations between objects as true objects, instead of using references which are clumsy to maintain (because the two or more ends in an association must be updated consistently).

The choice of a specific meta-model implementation is not really a problem, since our tool is capable of reading model description (and thus it can read the whole UML meta-model, which expresses itself in UML), and generating the adequate Eiffel code. The way to deal with associations is just a matter of specializing the generator.

But the Core Engine is more than a repository for elements in a model. Once a model has been loaded, it lives in memory as an Abstract Syntax Tree. Some utilities and tools are available to ease the building of plug-ins, by providing specific functions for the manipulation of this Abstract Syntax Tree. For example, it offers a hierarchy of *visitor design-patterns* which implement different traversal strategies of a model. A code generator is a specialization of an abstract `OO_CODE_GENERATOR` which overrides utilities methods such as *visit_ class* or *visit_ operation*.

3 A Framework for Automatic Transformation of UML models

A UML model consists of a large collection of modeling entities. In order to facilitate the transformation of such a model, we propose an object oriented framework that automates the tedious tasks involved with such a transformation. We propose the use of a mixed object-oriented and functional programming paradigm to develop a reusable toolbox of transformation operators. The functional paradigm has a strong orientation towards generic composition of operators while an object-oriented provides an intuitive extension mechanism via inheritance and aggregation. In general, transformation involves two distinct operations. First, the collection of meta-model elements in a given UML model needs to be traversed. During traversal, a given set of meta-model elements that conforms to a given criterion is selected and a transformation operator applied to it.

3.1 Iterating Model Elements

Each UML model is made up of an instance of a collection of meta-classes from the meta-model[9]. This meta-class collection forms a complex network of associations among one another. Among all these associations, we are particularly interested in the composition relation of these meta-model elements. This composition relation forms a spanning tree of all UML meta-model elements in the model. In the transformation framework, we apply an iterator over this spanning structure to produce a "linear" sequence of meta-model elements.

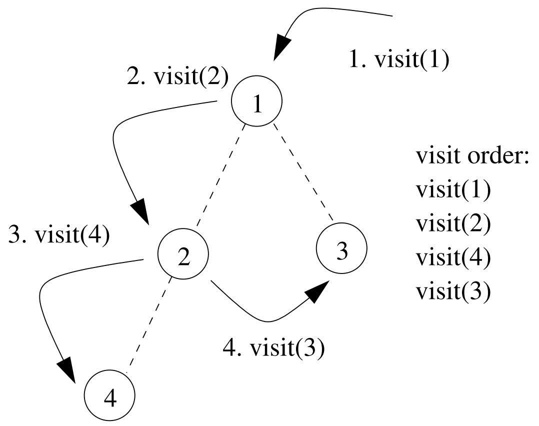

Figure 2: Depth-first traversal of Iterator

The "linearization" of the spanning tree allows us to apply standard list processing techniques for our transformation.

To understand how we "linearize" a spanning tree, it is necessary to know that our iterator implements the *Visitor* design pattern[7]. A typical iterator will "visit" a given root node of the meta-model spanning tree, which in turn request the iterator to "visit" its sub-nodes. This process is repeated recursively, completely traversing the spanning structure in depth-first order (see figure 2).

We designed our iterator to be driven externally, allowing our transformation operators to "see" the iterator as providing a "linear" sequence of elements. Once we have "virtually" linearized our meta-model instance, we can proceed with the description of the transformation operation itself.

3.2 Transformation Using An Applicative Approach

Our intention is to address the problem of providing a mechanism that separates the concerns of flexibly recombining transformation operations and their algorithmic details. We have identified three axes or aspects to this problem - Iteration of UML models, The manipulation operations, and the problem of flexibly and generically composing these operations. These goals are in line with making UMLAUT a powerful tool for manipulating UML models. A close look at the functional programming paradigm provides an interesting perspective to our problems.

In the context of the theory of lists[1], it has been shown that any operation can be expressed as the algebraic composition of a small number of polymorphic operations like *map*, *filter* and *reduce*. This idea has been exploited in the object oriented context by Pacherie in his thesis[17]. He propose to reify each of *map*, *filter* and *reduce* in the construction of a toolbox of algebraic operators for an object oriented framework for parallel computation[13].

We propose to extend these ideas to handle the object-oriented structures described by the UML meta-model.

In our transformation framework, the fundamental abstraction is a function mapping. We conceptualize a function fun as

$$fun : a \rightarrow b$$

which evaluates an object of type a to yield a result of type b. We can generically compose different functions as long as their type signatures match. I.e. given $f : a \rightarrow b$ and $g : c \rightarrow d$, we can compose g and f as in $g \circ f$ as long as the type of b matches the type of c. This lets us generically build complex transformation operations out of simpler primitives. It is independent of the details of what the operator does, or how it does it. The result is a framework of programming where a programmer deals only with the input parameter, the algorithm of the operation, and its result.

In the transformation of UML models, we will often consider collections of model elements. It would be practical to be able to apply the functional operators described previously on different types of collections while preserving the 'black box-ness' of the function. The approach we take is to use the *map* operator. It is a polymorphic operator that applies a function onto each element of a list and returns a list of result elements. The definitions of *map* is

$$map : (a \rightarrow b) \rightarrow ([a] \rightarrow [b])$$

where $[a]$ denotes a sequence of elements of type a, and $(a \rightarrow b)$ represents a function from a to b. Thus, we can view the application of *map* on a function yields a new function that works on sequences instead of singular entities. i.e. if $f : (a \rightarrow b)$ then $map\ f : ([a] \rightarrow [b])$. This abstraction works for any given function and preserves the generic composability described earlier for functions. Given these advantages, we implemented our *map* abstraction as in figure 3. The blocks each represent a specific functional abstraction. Applying *map* on f (i.e. f is an argument to the function *map*) yields a new composite function $map\ f$, as per definition of *map* above. The result is the polymorphic abstraction of 'apply f to every element in the list'. And it can be applied for any f and any list of elements of conforming types.

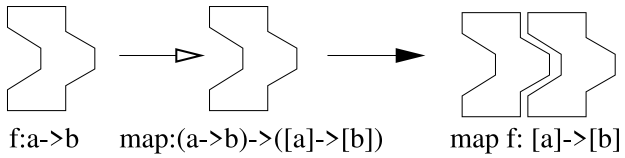

Figure 3: Map implementation

Similarly, we implement *filter* and *reduce* according to their definitions:

$$filter : (a \rightarrow boolean) \rightarrow ([a] \rightarrow [a])$$
$$reduce : (a \rightarrow a \rightarrow a) \rightarrow ([a] \rightarrow a)$$

Filter allows us to select elements based on a criterion and *reduce* helps us validate our model after transformation by collapsing the sequence into a single result.

3.3 Transformation Semantics

The transformation of a UML model can be summarized to consist of:

1. Addition of new elements to an existing model
2. Removal of model elements from an existing model
3. Modification of properties on an existing model element.

(1) and (2) are operations that modify the spanning tree structure of the UML model. As our iterator employs "lazy" traversal over this same structure, its modification during traversal presents a problem of ensuring "robust iteration". The use of "lazy" traversal is a trade-off between traversal efficiency and complexity.

(3), however, is an operation that yields no result. Its sole purpose is to produce an in-place update of model elements. Such an operation is widely known as "side-effect" in functional programming and we model operators belonging to this category using a *Void* return type. This hinders careless composition with a side-effect function. In summary, these two issues provide a strong motivation for further research on our transformation framework to derive a set of formal semantics for UML transformation operations.

4 Simulation and Validation of a UML model

4.1 A Distributed Multimedia Application Example

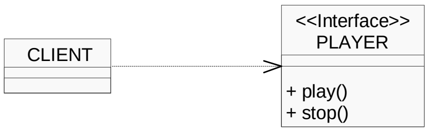

Figure 4: UML model of a media player

The example model is a video-on-demand application, shown in figure 4. `CLIENT` and `PLAYER` are remotely located and interact via a network.

The aim is to transform the initial UML model in figure 4 into an executable model representing the simulator, shown in figure 5. The new model contains abstract classes that represent reification of the concepts relevant to simulation (states, messages, timers) and classes representing the simulation engine that manipulates them. Those classes form a *validation framework*[12] that is configurable as per requirements. For example, to generate the `PLAYER_`*X*`_COMMAND`'s from the `PLAYER` class, we need to apply the following manipulations:

1. Given the sequence of all elements in our model, we isolate classes with the ≪interface≫ stereotype.

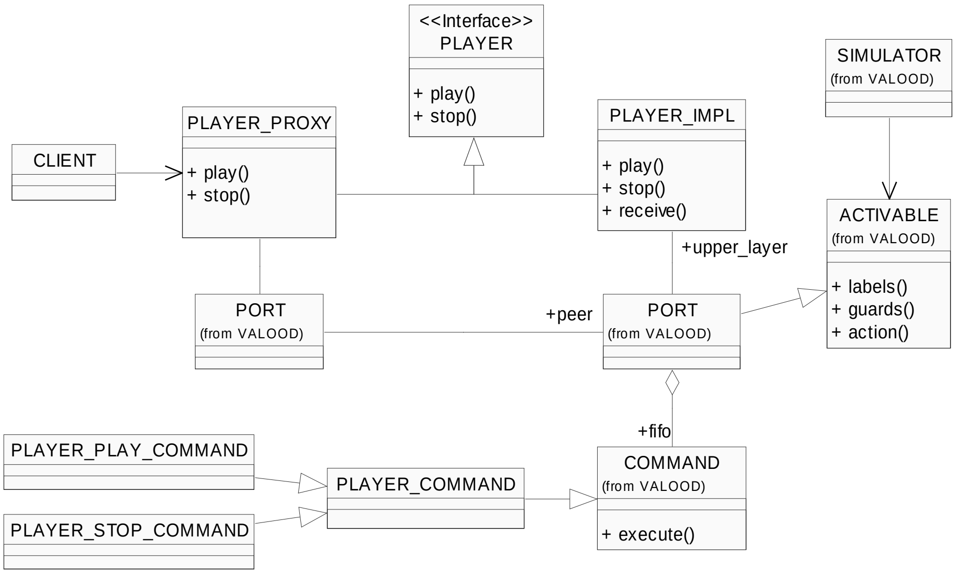

Figure 5: VALOODER validation framework

2. Extract the operations from this class. This results in a sequence of elements representing the class' operations.

3. Apply a COMMAND class generator over each operation for each class to produce the corresponding derived COMMAND class.

Using our operators, we will describe the transformation, T, as follow:
$T{=}(map\ (map\ makeCmdClass))\circ(map\ getOps)\circ(filter\ isIntfClass)$

Note that the last step involves a nested *map*. This is necessary because each class contains a set of operations. The functional abstractions from which we base our operators allow us to realize nested iterations simply by means of function application. The framework has virtually decoupled the concerns of iteration, operator composition and operator algorithm. It allows each aspect to be treated separately, giving a flexible programming structure. By composing different transformation blocks from our library of operators, we apply each incrementally over our original model of fig. 4 and we arrive at the final model of figure 5.

4.2 Accessibility graph of a UML model

The validation techniques we want to apply to UML models are based on Labeled Transition Systems (LTS). The accessibility graph of a model describes the evolution of a system in terms of states and transitions labeled by events (operation calls, timer expirations, message exchanges). The accessibility graph is seldom a finite graph, and so is not built exhaustively. Instead, it is explored progressively, as needed, starting from the initial state of the system (the root of the graph), then querying fireable transitions going out of a given state and choosing or discarding some of them following specific criterion.

4.3 The OPEN/CAESAR toolbox

Many tools such as model-checkers or test generators operate on the LTS formalism. They already have proven useful to validate LOTOS or SDL specifications. Adapting them for UML would provide UML users with an interesting choice of mature and robust validation tools. In this context, the OPEN/CAESAR toolbox[6] is particularly appealing. OPEN/CAESAR is a collection of validation tools based on a common interface offering services to build the accessibility graph of a specification. This interface is language independent, and several compilers are provided that compile specifications in LOTOS or SDL and make them available to validation tools through the standard graph library interface. Thanks to this separation of concerns, existing tools can be reused for a new language without change by implementing the corresponding compiler.

4.4 From UML models to simulation code

The transformation framework presented in section 3 is at the heart of the compiler that generates the simulation code for UML specifications. Basically, the compilation consists in transforming the initial UML model into an executable model representing the simulator. The new model contains abstract classes that represent reification of the concepts relevant to simulation (states, messages, timers) and classes representing the simulation engine that manipulates them. Those classes form a *validation framework*[12] that needs to be tailored for the particular model to be simulated. This is done by specialization of the framework's abstract classes. Transformations inspired by classical *design-patterns*[7] are used to refine the original model. For instance, the *State* design-pattern is used to implement objects' behavior, using specializations of the framework's STATE class. Similarly, messages exchanged among objects are reified as specializations of the framework's MESSAGE class, along the lines of the *Command* design-pattern.

4.5 Using validation tools on a UML model

Once the original model is immerged in the validation framework, UMLAUT's code generators provide an executable simulator conform to OPEN/CAESAR's graph library interface. OPEN/CAESAR's tools are then available to exercise the UML specification. Figure 6 shows OPEN/CAESAR's interactive simulation tools, in which the user can click on fireable transitions to drive the system.

Figure 7 has been obtained by OPEN/CAESAR's "generator" tool which builds the complete accessibility graph of a finite LTS. The UML specification represents a video-on-

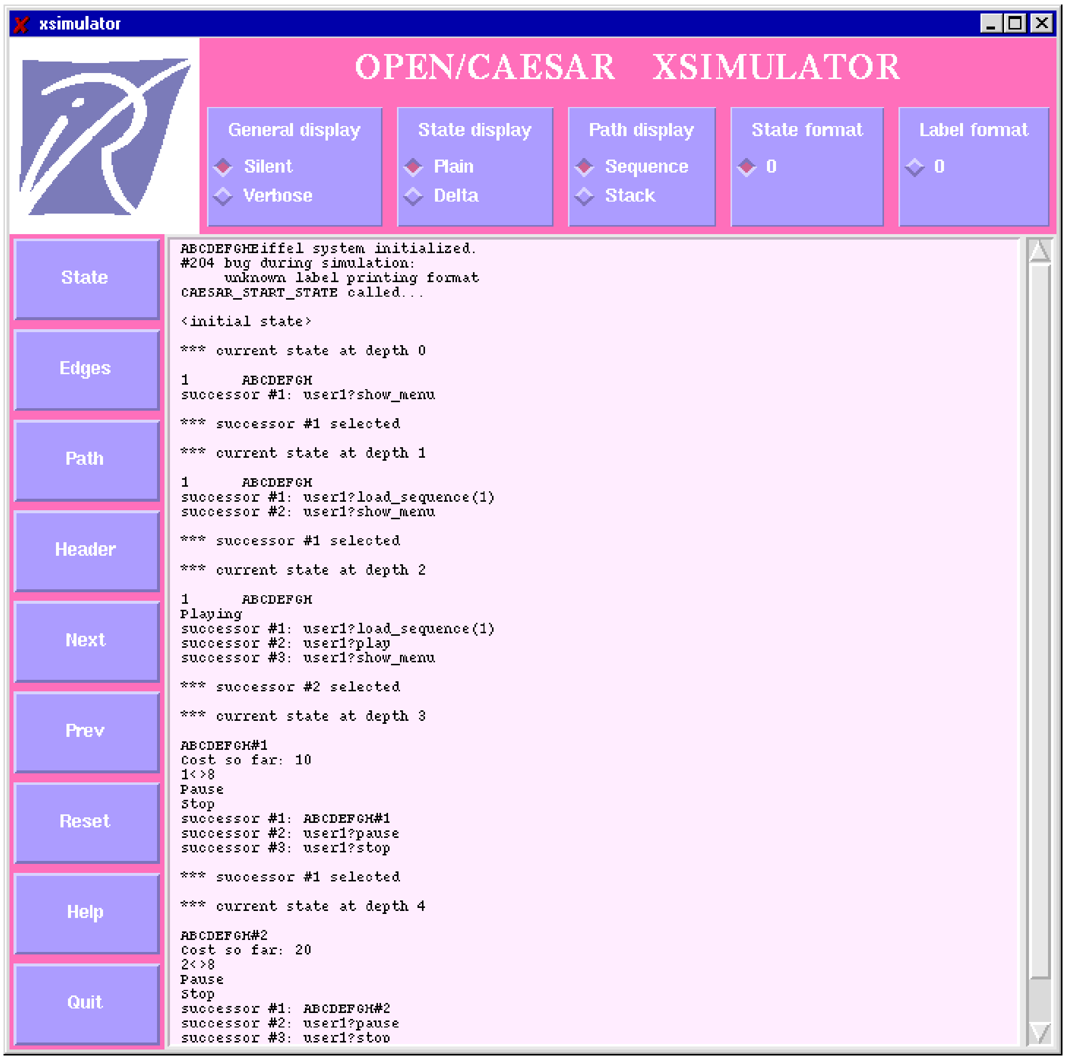

Figure 6: Interactive simulation (xsimulator)

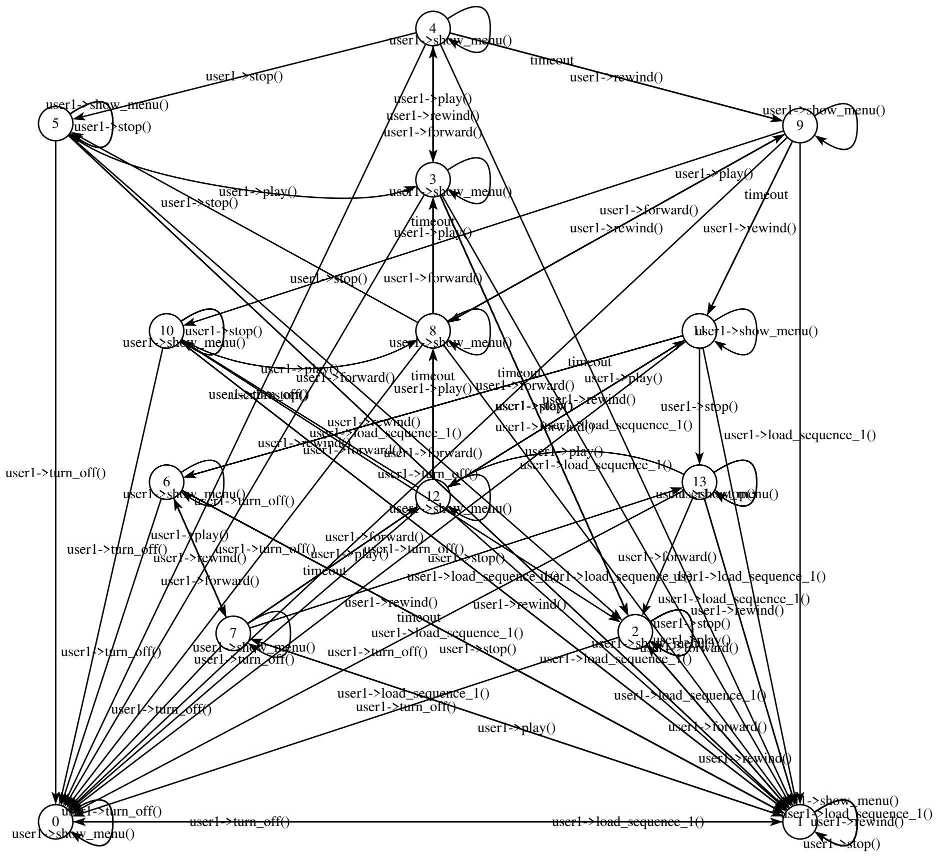

Figure 7: Accessibility graph of a simple UML specification

demand application (here with a single user watching a film that contains only a few frames, so that the graph is not too big.)

4.6 Future improvements of the UML simulator

Currently, only a subset of UML is taken into account by the simulator. Among the current limitations, we shall mention that only class diagrams and statecharts are accounted for in order to determine the behavior of the system. Moreover, statemachines communication is limited to asynchronous messages. Support for procedural nested flows of control is planned for a future release. We are actively working on extending support for the other behavioral views of UML models (collaborations, interactions, and activity graphs).

5 Related Work

Integrating the functional programming paradigm into an object-oriented context has been well studied by [2] and [16]. [3] also presents a graphical notation for visualizing functional composition. In particular, [17] and [14] show the increased versatility of iterators implemented in a functional manner.

With respect to UML model transformations, [4] propose the use of hypergenericity to describe model transformation. Hypergenericity is "the ability to automatically refine or transform a model by applying an external knowledge". This approach is supported by an object oriented interpreted language H[1] that allows the manipulation of UML model at the meta-model level. The constructs in H allow an expert to specify transformation rules that perform operations on the meta-model elements similar those of our proposed algebraic operators does.

A good source of reference for model transformation can be found in [8] where a set of equivalence rules for UML class diagrams and associations are presented. These rules can be integrated for model transformers prior to code generation because they express complex UML features using the basic core features that can be mapped directly to object oriented language constructs.

There are also a number of papers [15][5] that attempt to formalize some transformation rules on UML diagrams. We believe that model transformation in UML is a new subject of research and we believe that it is important to develop a set of general semantics that formalizes it. We hope to continue our work in using an applicative approach to address the formalism underlying model transformation and the semantics of the transformation operators.

[1] H is a language defined for manipulating a metamodel in the commercial CASE tool "Objecteering" by Softeam.

6 Conclusion

In this paper, we have outlined the functionalities and architecture of UMLAUT, a freely available UML Transformation Framework allowing complex manipulations to be applied to a UML model. These manipulations are expressed as algebraic compositions of reified elementary transformations. They are thus open to extensions through inheritance and aggregation. We have illustrated the interest of our approach by showing how the model of an UML distributed application can be automatically transformed into a labeled transition system validated using OPEN/CAESAR, a pre-existing protocol validation tool.

A preliminary version of UMLAUT is available on the web site of the UMLAUT project: http://www.irisa.fr/pampa/UMLAUT. Future work will be pursued in three directions: (1) to take into account UML more thoroughly, (2) to extend the transformation framework, (3) to make the UMLAUT software package more user-friendly and easier to use with mainstream UML modeling tools.

References

[1] R. S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springler-Verlag, 1987.

[2] Laurent Dami. *Software Composition: Towards an Integration of Functional and Object-Oriented Approaches.* Ph.D. thesis, University of Geneva, 1994.

[3] Laurent Dami and Didier Vallet. Higher-order functional composition in visual form. Object applications, Centre Universitaire d'Informatique, University of Geneva, August 1996.

[4] Philippe Desfray. Automation of design pattern: Concepts, tools and practices. In Pierre-Alain Muller and Jean Bézivin, editors, *Proceedings of UML'98 International Workshop, Mulhouse, France, June 3 - 4, 1998*, pages 107–114. ESSAIM, Mulhouse, France, 1998.

[5] Andy Evans. Reasoning with the Unified Modeling Language. In *Proc. Workshop on Industrial-Strength Formal Specification Techniques (WIFT'98)*, 1998.

[6] J-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and Sighineanu M. Cadp: a protocol validation and verification toolbox. In *Computer Aided Verification*, 1996.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.

[8] Martin Gogolla and Mark Richters. Equivalence rules for UML class diagrams. In Pierre-Alain Muller and Jean Bézivin, editors, *Proceedings of UML'98 International*

Workshop, Mulhouse, France, June 3 - 4, 1998, pages 87–96. ESSAIM, Mulhouse, France, 1998.

[9] Object Management Group. UML version 1.1, July 1997.

[10] Object Management Group. UML profile for enterprise distributed object computing (edoc) rfp, ad/99-03-10, 1999.

[11] Object Management Group. UML profile for scheduling, performance, and time rfp, ad/99-03-13, 1999.

[12] Jean-Marc Jézéquel, Alain Le Guennec, and François Pennaneac'h. Validating distributed software modeled with UML. In Pierre-Alain Muller and Jean Bézivin, editors, *Proceedings of UML'98 International Workshop, Mulhouse, France, June 3 - 4, 1998*, pages 331–340. ESSAIM, Mulhouse, France, 1998.

[13] Jean-Marc Jézéquel and Jean-Lin Pacherie. *Object-Oriented Application Frameworks*, chapter EPEE: A Framework for Supercomputing. John Wiley & Sons, New York, 1999.

[14] Thomas Kühne. Internal iteration externalized. In Rachid Guerraoui, editor, *ECOOP '99 — Object-Oriented Programming 13th European Conference, Lisbon Portugal*, volume 1628 of *Lecture Notes in Computer Science*, pages 329–350. Springer-Verlag, New York, N.Y., June 1999.

[15] Kevin Lano and Juan Bicarregui. Formalising the UML in structured temporal theories. In Haim Kilov and Bernhard Rumpe, editors, *Proceedings Second ECOOP Workshop on Precise Behavioral Semantics (with an Emphasis on OO Business Specifications)*, pages 105–121. Technische Universität München, TUM-I9813, 1998.

[16] Konstantin Laufer. A framework for higher-order functions in C++. In USENIX Association, editor, *Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS)*, pages 103–116, Berkeley, CA, USA, June 1995. USENIX.

[17] J.-L. Pacherie. *Système de motifs pour l'expression et la parallélisation des traitements d'énumérations dans un contexte de génie logiciel.* PhD thesis, IFSIC / Université de Rennes I, Décembre 1997.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399