UMLAUT: an Extendible UML Transformation Framework*

Wai Ming Ho Jean-Marc Jézéquel Alain Le Guennec François Pennaneac'h
IRISA/CNRS
Campus de Beaulieu
F-35042 Rennes Cedex, FRANCE
E-mail: {waimingh, jezequel, aleguenn, pennanea}@irisa.fr

Abstract

Advanced users often find themselves restricted by the limited facilities of most UML CASE tools when they want to do complex manipulations of UML model. E.g. apply design patterns, generate code for simulation and validation etc. In this paper, we describe UMLAUT, a freely available UML transformation framework, for manipulating UML models. 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.

1. Introduction

Since its standardization by the OMG in 1997, UML has become a *de-facto* standard modeling language for object-oriented systems and extensions have been proposed to address the needs of large critical systems. Its wide acceptance could be attributed to its expressive notation, open standard and support for distributed systems. However, the lack of rigorous formalization in UML hinders its application on complex distributed systems. On the contrary, formal techniques like model-checking, simulation and test generation are well adapted in this domain[10]. Unfortunately, formal techniques introduce a steep learning curve and usually impose modeling restrictions on the user[11].

We advocate combining the ease of use of UML and the strength of formal techniques to ensure reliability in the development of complex concurrent systems. We propose a UML model transformation tool, UMLAUT, that allows us to transform UML models formally and connect them to existing FDT tools. In section 2, we describe the overall architecture of UMLAUT, and section 3 presents the transformation framework that allows UMLAUT to transform a UML model. Finally, section 4 demonstrates how we transform an application for a simulator operating on the labeled transition system (LTS) formalism using UMLAUT. We conclude in section 5 with a discussion on related work.

*This work has been partially funded by CNET under the METAFOR project.

2. UMLAUT : An Extendible Transformation Framework

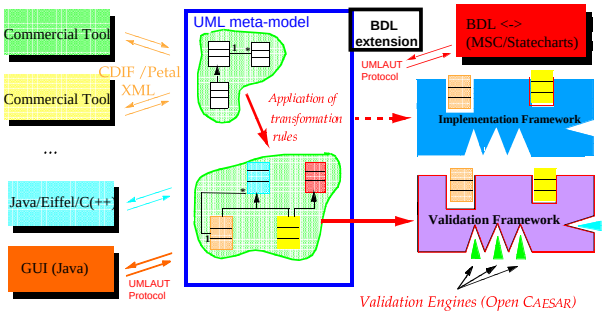

Figure 1. UMLAUT architecture

UMLAUT is a tool dedicated to the manipulation of UML models. It consists of a core engine which communicates with its surroundings via *hot-spots* (i.e. interfaces), where functional modules can be plugged in to specialize the behavior to meet specific requirements (see figure 1). Existing modules include code-generators for Eiffel and Java, import facilities for CDIF and Rational Rose[1] MDL files, reverse engineering from Java and Eiffel source, pluggable transformation models dedicated to the validation of distributed reactive systems, and communication with a Java graphical user interface.

[1]Rose98 is a trademark of Rational Software Corporation.

The UMLAUT Core Engine implements the UML meta-model, as described in [9]. Its implementation in the Eiffel programming language is a direct mapping of each UML meta-class onto an Eiffel class. The UML model is represented in memory as an Abstract Syntax Tree and the base implementation offers a hierarchy of *visitor design-patterns*[7] which implement different traversal strategies of the loaded model. E.g. the Eiffel and CDIF code generators provided by UMLAUT use this pattern.

3. The UMLAUT Transformation Framework

UMLAUT's transformation framework is designed using a mix object-oriented and functional programming paradigm. The transformation architecture models the computation concepts of functional operators so that transformation operators can be re-composed flexibly. Object-oriented inheritance and aggregation provide a mechanism for extending these operators. Transformation of UML models involves traversing the UML meta-model instance and applying the transformation operators on each element during the traversal.

3.1. Traversing a UML model

Each UML model is made up of a collection of meta-classes instances of the meta-model[9]. This meta-class collection forms a complex network of associations among one another. Of particular interest are the composite aggregate relationships of these meta-model elements. They form a spanning tree of all UML meta-model elements that describe the model. In our iterator, we traverse this spanning tree and conceptually "linearize" the meta-model element sequence. The interest of presenting a model as a "sequence" of model elements is the ability to apply standard list processing techniques on these elements.

3.2. Transformation using an applicative approach

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*. In [16], Pacherie presents an implementation in the object-oriented context where each of *map*, *filter* and *reduce* is reified as Eiffel classes. It forms the core of a toolbox of algebraic operators for an object-oriented framework for parallel computation[12]. We propose to extend these ideas to develop a reusable toolbox of transformation operators for 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.

Given the definitions of *map* as

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

where $[a]$ denotes a sequence of elements of type a, we implement it as in figure 2. 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 resulting $map\ f$ is an operator capable of applying f on each element of our "linearized" model. The implementation is related to higher order calculus in functional programming.

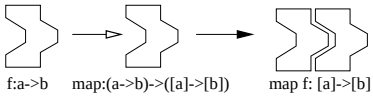

Figure 2. Map implementation

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

The validation techniques we want to apply to our 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 criteria.

For our example, we use the OPEN/CAESAR toolbox[6] to produce the accessibility graph of a UML model. 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 allows UMLAUT to generate an executable simulator conforming to this graph library interface.

4.1. From UML models to simulation code

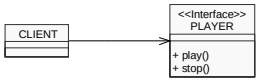

Figure 3. UML model of a media player

The example model is a video-on-demand application, shown in figure 3. `CLIENT` and `PLAYER` are remotely located and interact via a network.

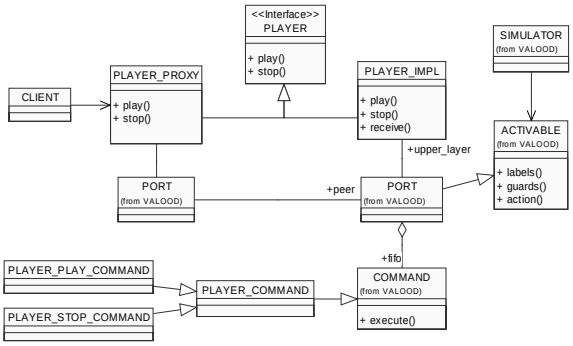

Figure 4. VALOODER validation framework

The aim is to transform the initial UML model in figure 3 into an executable model representing the simulator, shown in figure 4. 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*[11] that is configurable as per requirements. To demonstrate our transformation operators, we show how to generate the `PLAYER_`X`_COMMAND`'s from the `PLAYER` class. Let p be a predicate that determines if a model element is an operation of the class `PLAYER` and g be an operator that creates a class from an operation model element. We then compose our transformation operator, T, as

$$T = map\, g \circ filter\, p$$

Applying T to an iterator of our initial model will produce the `PLAYER_PLAY_COMMAND` and `PLAYER_STOP_COMMAND` from the operations `play` and `stop` of `PLAYER`. Such operations are repeated for each distinct transformation until we arrive at the final model of figure 4.

4.2. 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 that conforms to OPEN/CAESAR's graph library interface. Figure 5 shows an accessibility graph generated from an OPEN/CAESAR interactive tool. It builds the complete accessibility graph of a finite LTS of a single user watching a film that contains only a few frames.

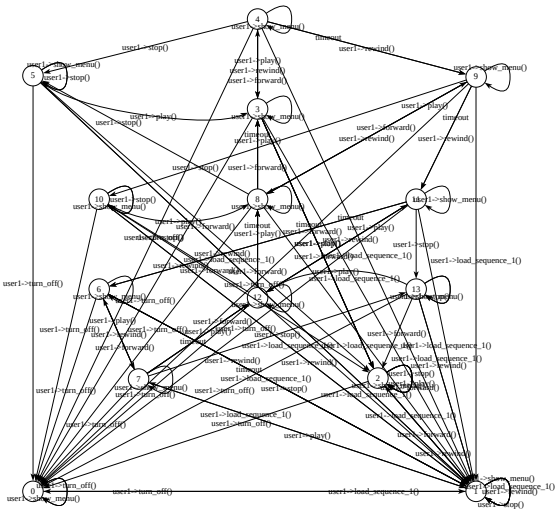

Figure 5. Accessibility graph of a simple UML specification

4.3. 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, statecharts and deployment diagrams 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 working on extending support for 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 [15]. [3] also presents a graphical notation for visualizing functional composition. In particular, [16] and [13] show the increased versatility of iterators implemented in a functional manner.

With respect to UML model transformations, [4] propose the use of hypergenericity. Hypergenericity is "the ability to automatically refine or transform a model by applying an external knowledge". This is supported by an interpreted object-oriented language H[2] that supports manipulation of UML models at the meta-model level. [8] outline several equivalence rules on model transformation that can be integrated to code generation because they express complex UML features using a subset of basic core features that can be mapped directly to object-oriented language constructs. Finally [14] and [5] give some examples of formalizing transformation rules on UML diagrams.

6. Conclusion

In this paper, we have introduced the functionalities and architecture of UMLAUT. We showed how a UML model of 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 and formalize the semantics of transformation, (3) to make the UMLAUT software package more user-friendly and easier to use. E.g. as a plug-in for mainstream UML modeling tools.

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

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

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

[4] P. Desfray. Automation of design pattern: Concepts, tools and practices. In P.-A. Muller and J. Bézivin, editors, *Proceedings of UML'98 International Workshop*, pages 107–114. ESSAIM, Mulhouse, France, 1998.

[5] A. 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 S. M. Cadp: a protocol validation and verification toolbox. In *Computer Aided Verification*, 1996.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[8] M. Gogolla and M. Richters. Equivalence rules for UML class diagrams. In P.-A. Muller and J. Bézivin, editors, *Proceeding of UML'98 International Workshop*, pages 87–96. ESSAIM, Mulhouse, France, 1998.

[9] O. M. Group. UML version 1.1, July 1997.

[10] J.-M. Jézéquel. Experience in validating protocol integration using Estelle. In *Proc. of the Third International Conference on Formal Description Techniques, Madrid, Spain*, November 1990.

[11] J.-M. Jézéquel, A. L. Guennec, and F. Pennaneac'h. Validating distributed software modeled with UML. In P.-A. Muller and J. Bézivin, editors, *Proceedings of UML'98 International Workshop*, pages 331–340. ESSAIM, Mulhouse, France, 1998.

[12] J.-M. Jézéquel and J.-L. Pacherie. *Object-Oriented Application Frameworks*, chapter EPEE: A Framework for Supercomputing. John Wiley & Sons, New York, 1998.

[13] T. Kühne. Internal iteration externalized. In R. 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.

[14] K. Lano and A. Evans. Rigorous development in uml. In *Joint European Conference on Theory and Practice of Software – ETAPS '99*, volume 1577 of *LNCS*. Springer, mar 1999.

[15] K. 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.

[16] 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.