RIGOROUSLY AUTOMATING TRANSFORMATIONS OF UML BEHAVIOR MODELS

Jon Whittle[1], João Araújo[2], Ambrosio Toval[3], and Jose Luis Fernández Alemán[3]

[1]QSS / NASA Ames Research Center, M/S 269-2 , Moffett Field, CA 94035
USA, TEL:+650-604-3589
jonathw@ptolemy.arc.nasa.gov
[2]Departamento de Informática, Faculdade de Ciências e Tecnologia,
Universidade Nova de Lisboa, 2825 Monte da Caparica, PORTUGAL,
TEL: + 351-1-2948536; FAX: + 351-1-2948541
ja@di.fct.unl.pt
[3]Software Engineering Research Group, Department of Informatics,
University of Murcia, SPAIN, TEL: +34-968-364621
aleman@dif.um.es, atoval@um.es

Abstract. UML is a standard modeling language that enables the specification of applications at many different levels of abstraction using a wide range of notations. However, there is currently only limited research investigating the maintenance of UML models throughout the software lifecycle. In particular, UML behavior models are typically not orthogonal but each model shows similar behaviors from different perspectives. Hence, modifications to one behavior model will have possibly unforeseen ramifications elsewhere. We propose a framework in which to express automated transformations on UML models using a formalization of UML in rewriting logic. These transformations include abstractions and refinements of behavior models as well as translations between models. What they have in common is that they take into account the global behavior of the system, expressed over multiple models, rather than being localized to a particular notation.

1 Introduction

The Unified Modeling Language (UML) [4] provides several concepts and respective notations to be used at different levels of abstraction throughout the development process. For example, in a use case-driven approach, as proposed by Jacobson [8], use cases describe functional requirements of a system at a very high level of abstraction. Use cases can then be refined by giving collaboration or sequence diagrams that help to identify the set of user requirements. Collaborations express global behavior regarding the interaction between objects. Hence, for refinement to the implementation level, the designer must provide local behavioral models, e.g., statecharts, from which code can be developed. In an ideal world, each development phase is completed before progressing to a less abstract layer. In practice, however, the development of UML models is highly iterative. This poses a problem since UML

models are not independent of each other, and so any modifications in a particular model must be reflected in related models.

Updating UML models is a highly time-consuming and error-prone task if done manually. Even the slightest modifications can have far-reaching ramifications on the rest of the model, which may go unnoticed by designers. We propose automated tool support for transforming UML models, where the transformations may be minor model updates, abstractions, refinements, or translations between notations. The aim is to equip the designer with a suite of transformations that make some specified change whilst maintaining the consistency of the system as a whole. Such consistency maintenance is only possible given a formalization of the model semantics. We use rewriting logic, and its implementation in Maude [5], as a vehicle to express transformations and their properties.

2 Formal Framework

Rewriting logic is a very flexible reflective logic that has very good properties as a logical and semantic framework. It can be interpreted logically or computationally, the latter interpretation giving rise to an executable specification language implemented as the Maude system. It is a logic of concurrent change that can deal naturally with state and with highly nondeterministic concurrent computations. In particular, it supports concurrent object-oriented computation. These properties of rewriting logic make it an ideal framework in which to formalize UML. Moreover, since Maude is based on conditional rewrite rules, it is very natural to express transformations of UML models.

A rewrite theory is a pair *(T,R)* where *T* is an equational theory and *R* is a collection of labeled and possibly conditional rewrite rules involving terms in the signature of *T*. Rewrite rules are of the form $r: t \rightarrow t'$ and can be applied modulo associativity, commutativity, identity and idempotency axioms. This leads to a large number of possible rewriting paths which can be controlled by strategies implemented using Maude's reflective capabilities.

Alemán & Toval [1] shows how Maude can be used to formalize UML class diagrams. A similar approach can be used for UML behavior models. As an example, the UML statechart shown in Figure 3 is formally specified by a pair *(transitions, hierarchy)* where *transitions* denotes a list of transitions between states, and *hierarchy* represents a state hierarchy (e.g. see formalization in Figure 1). These formal terms are expressed according to the existing UML Statechart formal specification [1] at the UML metamodel layer. Once the statechart is formally represented, it can be mathematically manipulated and prototyped. Likewise, rigorous transformations can also be applied. So far, class diagrams, statecharts, and a subset of OCL have been formalized as Maude models.

```
transitions1=
    transition (initialState, s1, empty)          transition (s2, xTrue, m1)
    transition (initialState, s5 empty)           transition (s3, s5, m2)
    transition (initialState, xFalse, empty)      transition (s3, s4, m3)
    transition (s1, s2, m2)                       transition (s4, s5, m2)
    transition (s1, s3, m1)

hierarchy1 =
OrState (ST, empty)
OrState (xFalse, empty, simpleState (s1, empty) simpleState (s2, empty))
OrState(xTrue, empty, simpleState (s3, empty) simpleState (s4, empty) simpleState(s5,empty))
```

Fig. 1. Maude formalization of statechart in Figure 3

3 Transformations

As previously mentioned, Maude provides a suitable framework for expressing both the semantics of UML behavior models and all kinds of transformations over these models. Aleman, Toval & Hoyos [2] and Lano & Bicarregui [7] identify a number of different possible categories of transformations. Rather than try to cover all possible transformations here, we prefer to give a small number of illustrative examples. Ultimately, we expect these examples to be a subset of a much larger suite of transformations that will be made available to the UML designer. Our intention is to further refine our formalization of UML in Maude and implement transformations in Maude, such as those given in this section.

3.1 Example one: refinement from requirements to design

This section gives an example of a very large grain transformation. In the use-case driven approach to UML development, use cases are described using collaborations which express a global view of expected traces of message-passing between objects. In the initial design stages, each class is specified locally using, for example, a statechart. This is a local view of the behavior of the system. Since collaborations are requirements traces of behavior, they should remain true in all refinements of the statechart models. Currently, however, there is very little automated support for providing this guarantee. The situation is complicated further by the fact that there may be additional constraints on the system behavior, usually expressed using OCL. We have developed a transformation that can translate a collection of collaborations (in our case, sequence diagrams) annotated with OCL pre- and post-conditions on messages into a collection of statecharts. This should be an invaluable translation allowing a designer to easily update his/her statechart models as the requirements change. Note that this translation requires reasoning over the semantics of the model, since the OCL constraints express semantic restrictions on the way that messages can be ordered. In particular, different sequence diagrams may be merged/interleaved

during the translation, or they may conflict with each other, in which case the user is notified. Whittle & Schumann [9] gives a full description of the translation, currently implemented in Java. Our intention is to integrate the transformation into the framework of rewriting logic.

The key idea in the transformation is to define the concept of state as the values of *state variables*, taken from the OCL constraints, along with information about the ordering of messages, taken from the sequence diagrams. Using this idea, we can associate each node in each statechart with a *state vector*, giving state variable values. Some nodes may have the same state vector but are separated by transitions that directly correspond to messages passed in a sequence diagram. Figure 2 gives an overview of the process.

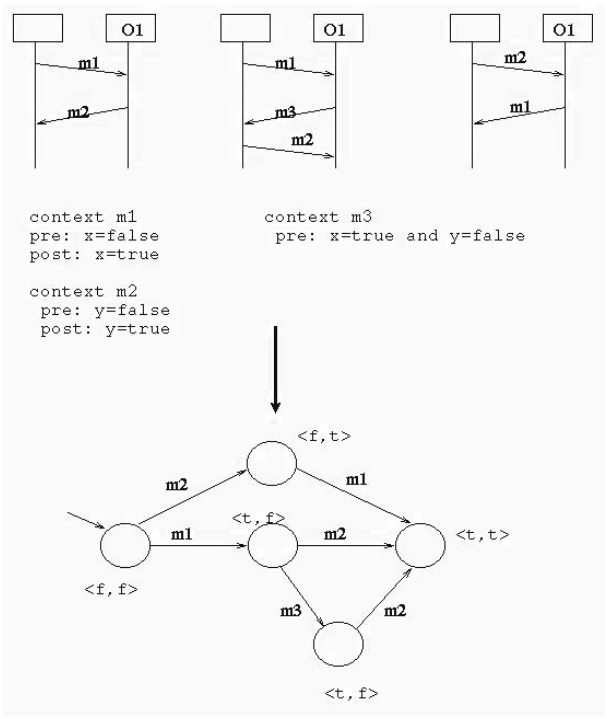

Fig. 2. From sequence diagrams to statecharts

Messages involving an object, *O*, in a sequence diagram contribute to the behavior of the class to which *O* belongs. The statechart shown in the figure is for the class containing *O1* based on the three sequence diagrams and the given constraints. Note how we characterize each statechart node using the state vector <*a*,*b*> where *a* and *b* are the values of the state variables *x* and *y*, respectively. This leads to a much more informed merging of sequence diagrams, resulting in fewer nodes and taking into account background knowledge given in OCL. The transformation itself consists of deriving and propagating sets of state variable values using unification (which suggests loops in the statechart) and frame axioms.

3.2 Example two: reverse engineering

An example of a transformation preserving behavioral correctness would be one that made only structural changes on the behavioral model, e.g., one that introduced hierarchy / orthogonality into a statechart. This kind of transformation would be important to introduce readability into a model before it was passed to another designer, or to re-engineer an existing model. Hierarchy can be introduced automatically by using background knowledge available in the form of OCL constraints. Consider the statechart in Figure 2 again. As before, the OCL constraints can be used to characterize the nodes by assigning a state vector to each. If the variables in the state vector correspond to "modes" of the system, then it is natural to partition the statechart over the values of these variables. Suppose, for instance, that variable *x* specifies whether or not a machine is switched on. Then partitioning over the value of *x* gives the hierarchical statechart in Figure 3, which corresponds to the kind of structure a designer might introduce.

Note that a partitioning algorithm could be applied recursively to introduce multiple layers into the statechart. In general, there should be heuristics built into the transformation to ensure a "good" structure: e.g., the hierarchy should not be too deep, minimize the number of inter-level transitions. In addition, only a subset of the state variables will correspond to modes and so the designer should be given the ability to express which variables are *mode variables*.

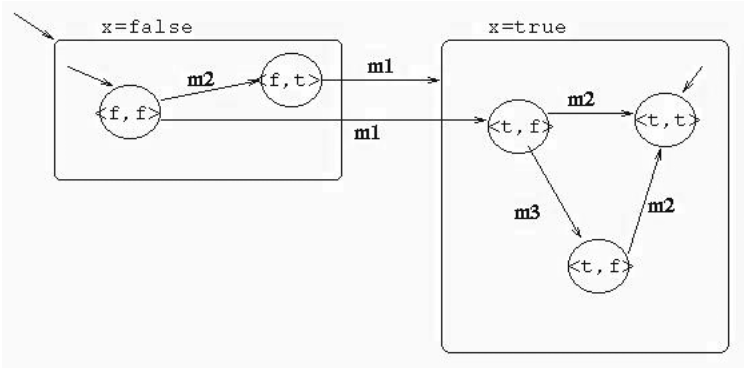

Fig. 3. Partitioning over mode variables

3.3 Example three: refinement at the same abstraction level

We will also consider refinements similar to Opdyke's refactorings [6]. For example, Figure 3 shows the result of applying a transformation to abstract an attribute and a method from a class into a new superclass, and the effect that this has on the corresponding statecharts. This example assumes an interpretation in which each class has its own statechart describing (a subset of) the behavior of that class. Class *B* is refined into an inheritance hierarchy in which class *A* is the parent of *B*. The attribute *x* and the method *a* are moved to class *A*. Note that the statechart for *B* will remain the same (as no behavior has been added or removed), but a statechart for

A can be inferred if there are also OCL constraints on the transitions which allow state vectors to be assigned in the usual way.

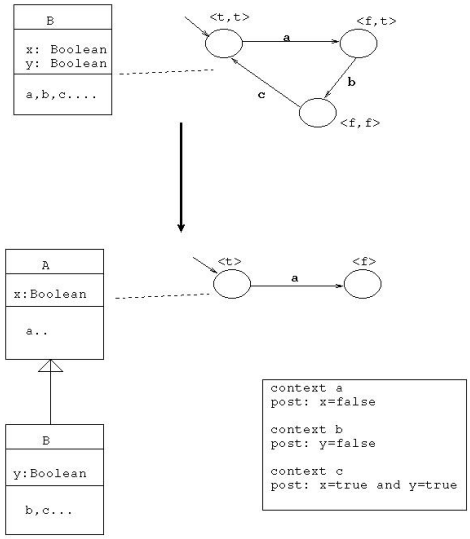

Fig. 4. Introducing class inheritance.

4 Related Work and Conclusions

The idea of coming up with UML behavior model transformations is not new. Indeed, Lano & Bicareggui [7] describes a number of examples of such transformations. However, the particular kinds of transformations that we are considering are novel. First, they are not necessarily correctness-preserving. Second, they include large grain transformations which do much more than make minor modifications. It is mostly only small grain transformations that have been studied previously. Small grain transformations alone are not sufficient. To make any useful change would require a long sequence of such transformations. Coming up with the right sequence is a difficult task and can be automated by our large grain translations. Moreover, most previous transformations work only on static UML models (e.g. class diagrams).

The other novelty in our approach is the use of Maude and rewriting logic. Lano & Bicareggui uses a real-time action logic (RAL) and proves simple properties of the transformations presented there. However, there are no automated proof tools for RAL and so all proofs had to be done manually. The use of Maude allows proofs to be developed automatically at design time, which gives much greater flexibility than if all proofs had to be done by hand off-line. We have developed a way of formalizing collaborations using Object-Z [3], but once again, there is only limited proof support for Object-Z.

References

1. Alemán, J.L.F., Toval, A.: Formally Modeling and Executing the UML Class Diagram. In Rodriguez, M.J., Paderewski, P. (eds.): Proc. of the V Workshop MENHIR (Models, Environments, and Tools for Requirements Engineering), Universidad de Granada, Spain (March 2000).
2. Alemán, J.L.F., Toval, A., Hoyos, J.R.: Rigorously Transforming UML Class Diagrams. In Rodriguez, M.J., Paderewski, P. (eds.): Proc. of the V Workshop MENHIR (Models, Environments, and Tools for Requirements Engineering), Universidad de Granada, Spain (March 2000).
3. Araújo, J., Moreira, A.: Specifying the Behaviour of UML Collaborations Using Object-Z. In Proc. of the Americas Conference on Information Systems, Long Beach, California (August 2000).
4. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide, Addison-Wesley, Reading, Massachusetts (1998).
5. Clavel, M., Eker, S., Lincoln, P., Meseguer, J.: Principles of Maude. In Proc. of the 1st International Workshop on Rewriting Logic and its Applications (1996).
6. Fowler, M.: Refactoring: improving the design of existing code. Addison Wesley, Reading, Massachusetts (1999).
7. Lano, K.C., Bicarregui, J.C.: UML Refinement and Abstraction Transformations. In ROOM 2 Workshop, Bradford University (1998).
8. Jacobson, I.: Object-Oriented Software Engineering - a Use Case Driven Approach, Addison-Wesley, Reading Massachusetts (1992).
9. Whittle, J., Schumann, J.: Generating Statechart Designs from Scenarios. In Proc. of International Conference on Software Engineering (ICSE2000). Limerick, Ireland (2000).