Automatic Synthesis of UML Designs from Requirements in an Iterative Process

Johann Schumann[1] and Jon Whittle[2]

[1] RIACS/NASA-Ames
[2] QSS/NASA Ames
email:schumann,jonathw@ptolemy.arc.nasa.gov

The Unified Modeling Language (UML) is gaining wide popularity for the design of object-oriented systems. UML [6] combines various object-oriented graphical design notations under one common framework. A major factor for the broad acceptance of UML is that it can be conveniently used in a highly iterative, Use Case (or scenario-based) process (although the process is not a part of UML). Here, the (pre-)requirements for the software are specified rather informally as Use Cases and a set of scenarios. A scenario can be seen as an individual trace of a software artifact. Besides first sketches of a class diagram to illustrate the static system breakdown, scenarios are a favorite way of communication with the customer, because scenarios describe concrete interactions between entities and are thus easy to understand. Scenarios with a high level of detail are often expressed as sequence diagrams.

Later in the design and implementation stage (elaboration and implementation phases), a design of the system's behavior is often developed as a set of statecharts. From there (and the full-fledged class diagram), actual code development is started. Current commercial UML tools support this phase by providing code generators for class diagrams and statecharts.

In practice, it can be observed that the transition from requirements to design to code is a highly iterative process. This means that initial versions of requirements have to be modified and refined to meet additional (customer) wishes and constraints. Also modifications of the code can lead to revisions in design. This iterative behavior is strongly supported by most modern processes, because it facilitates early detection of inconsistencies and bugs. Fixing a bug which is detected late in the software lifecycle can cost approximately 60-100 times more than one which is detected early [3].

However, current UML tools do not support the transition from requirements to design in a comfortable and consistent way. Often, a considerable amount of time is spent to write down the requirements in great detail. Then the requirements tend to be "forgotten" until test cases have to be set up. At this point of time, it is usually detected that those requirements are hopelessly out of date and require a major overhaul.

Our work [7] addresses these issues and tries to close the gap between requirements and design. In this talk, we present a set of algorithms which perform reasonable synthesis and transformations between different UML notations (sequence diagrams, OCL constraints, statecharts). Our overall aim with respect to reasonable synthesis is centered around the following concepts: detection of inconsistencies and ambiguities in sequence diagrams, merging of similar or duplicated behaviors from different sequence diagrams, the production of highly readable (structured) statechart, and the support for iterative refinements. More specifically, we will discuss the following transformations.

Statechart synthesis. From a set of sequence diagrams with object O (as an instance of a class C) as a participant, we automatically synthesize a statechart which reflects C's behavior given in the sequence diagrams. Because the standard semantics of sequence diagrams is very weak, almost no duplicate or similar behavior can be merged. In order to overcome this problem, we allow the designer to specify a set of OCL constraints, describing pre- and postconditions over a vector of "state-variables" for messages in the sequence diagrams. These state-variables (currently of type boolean) and the constraints are used by our algorithm to detect conflicts between a sequence diagram and the OCL constraints (the domain model) using unification and a version of the frame axiom. Furthermore, potential loops can be detected. Our state variables also form the basis for constructing the (flat) statechart. In contrast to other approaches (e.g., that used in the SCED tool [2]), the domain model allows a justified merge of sequence diagrams. Because OCL constraints need to be defined only for few (possibly important or ambiguous) messages, we believe that the additional burden for the designer is kept to a reasonable level.

Introduction of hierarchy. As soon as the design gets more complex (i.e., a statechart contains more than approx. 5 nodes), things usually get out of hand, because the design cannot be read by the designer/developer in a reasonable manner. D. Harel [1] tackled this problem by introducing hierarchy and orthogonality in his statecharts. Nodes can be grouped into supernodes, increasing readability and avoiding an explosion of states when new functionality is added.

In order to produce useful designs, our algorithm is capable of synthesizing hierarchical statecharts. Thereby, the initial flat statechart is partitioned recursively according to a given strategy, usually based upon information in the class diagram, a given ordering of the state-variables, and user preferences. Because hierarchy is transparent with respect to statechart semantics, multiple different hierarchies (or "views") can exist in the system at the same time.

Consistency of modifications. In most software projects, requirements scenarios only cover a (hopefully important) fragment of the intended system behavior. Therefore, the synthesized statechart can only be a first design sketch which needs to be generalized and modified by the designer. A hierarchical structure (see above) is an important prerequisite for such activities. However, transformations or modifications easily can invalidate the requirements. Therefore, we have developed a "backwards direction" algorithm which checks consistency of the modified statechart with the original requirements and the domain model. In case an original sequence diagram has been violated, our algorithm proposes a set of revised (added/modified/deleted messages) according to given criteria.

"Design-Debugging". Despite the well-known "fact" that every programmer always writes error-free code, debugging of a software artifact is an extremely important (and unfortunately time-consuming and costly) task. Our algorithms support debugging of UML diagrams on various levels [5]. Early checking of consistency in the requirements is one way of debugging during very early stages of the development, i.e., already before the actual design starts. Our backwards-direction algorithm facilitates finding bugs in modifications of the original design. Here, the user is not required to manually go through (lengthy) execution traces. All the user has to do is to check the proposed modifications (which are usually much smaller) of the sequence diagrams whether or not they are consistent with the intended system behavior.

A popular method for debugging is the so-called "printf-debugging". Here, the programmer instruments the code with statements which write trace information and variable values into a log-file. After the program execution, the trace in analyzed. In practice, however, annotation of larger program to detect a certain behavior is far from trivial. Usually, a lot of distant and seemingly unrelated parts of the code have to be annotated. Here, our algorithm for the introduction of hierarchy can be of great help. Combined with the automatic code generation facilities of commercial UML tools, such an instrumentation can be accomplished easily. The developer changes the hierarchy of the statechart(s) in such a way that all states which are of interest for the current debugging session are grouped together in one (or a few) superstates on the top of the hierarchy. Then, all important parts are clearly visible and can be instrumented easily (e.g., by adding specific debugging actions). The change of the hierarchy can be initiated by giving additional constraints over the state variables.

Our entire set of algorithms is based upon a logic-based semantics of the different UML notations. We are currently only using a subset of the sequence diagram and statechart notation, for which there is a straightforward, undisputed semantics. In future, we will work on the incorporation of additional elements of the statechart notation and extensions of sequence diagrams (see [8] for details) into our framework.

We have developed a prototype of these algorithms in Java. Integration into a UML tool (using XMI) is currently in progress. We have tried out our algorithm with various small examples, like the ATM machine and a cruise-control system. Future work includes NASA-internal case studies on space shuttle software and software for advanced air traffic control.

However, there is much work still to be done. Our overall goal is to have an integrated UML support tool which is concise and accurate, but hides the underlying formal techniques (unification, constraint solving, tree searches) as much as possible. By integration of the algorithms into commercial UML tools we aim at "invisible formal methods" as proposed by J. Rushby [4]. The incorporation of additional domain information in the form of OCL constraints allows concise consistency checks and justified merging of sequence diagrams with minimal overhead for the software designer and developer. It is thus expected that such tools will increase productivity and quality of object oriented software systems.

References

1. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
2. T. Männistö, T. Systä, and J. Tuomi. SCED report and user manual. Report A-1994-5, Dept of Computer Science, University of Tampere, 1994.
3. R. Pressman. *Software Engineering - a Practitioner's Approach*. McGraw-Hill, 1997.
4. J. Rushby. Disappearing formal methods. In *Proceedings of HASE: Fifth IEEE International Symposium on High Assurance Systems Engineering*, 2000. invited paper.
5. J. Schumann. Automatic debugging support for UML designs. In M. Ducasse, editor, *Proceedings of the Fourth International Workshop on Automated Debugging*, 2000. http://xxx.lanl.gov/abs/cs.SE/0011017.
6. Unified Modeling Language Specification, Version 1.3, 1999. Available from Rational Software Corporation, Cupertino, CA.
7. J. Whittle and J. Schumann. Generating Statechart Designs From Scenarios. In *Proceedings of International Conference on Software Engineeering (ICSE 2000)*, pages 314–323, 2000.
8. J. Whittle and J. Schumann. Generating Statechart Designs From Scenarios. *TOSEM*, 2001. submitted.

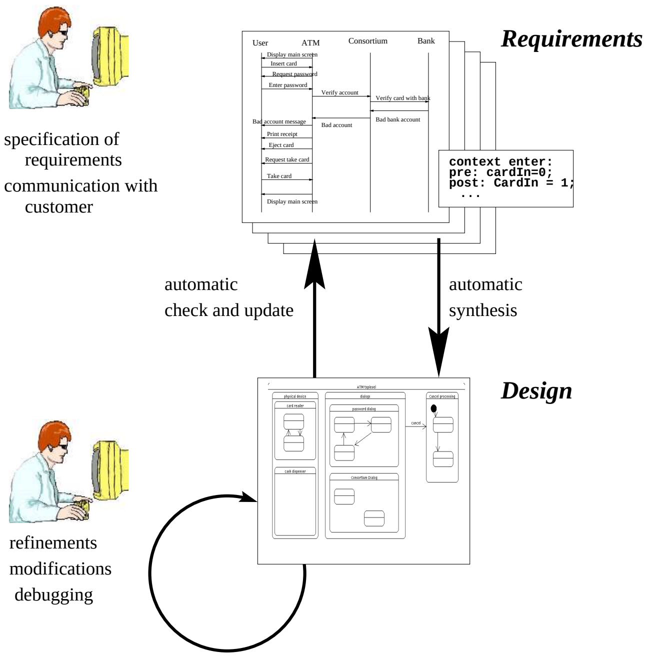

Fig. 1. Automatic synthesis of statecharts in a highly iterative software process