

Scenario-based Synthesis of Annotated Class Diagrams in UML

Petri Selonen and Tarja Systä

Tampere University of Technology, Software Systems Laboratory,
P.O.Box 553, FIN-33101 Tampere, Finland

{pselonen,tsysta}@cs.tut.fi

Abstract. This paper discusses how to generate a class diagram, annotated with operation descriptions, from a set of sequence diagrams in the Unified Modeling Language (UML). The transformation process consists of the following steps. First, we translate the elements in a sequence diagram to elements of class diagrams using basic transformation rules together with more advanced heuristics. We then use an existing technique for synthesizing state machines from a set of sequence diagrams to describe an overall structure of individual operations. Finally, we transform the synthesized state machines to pseudocode presentations, which are attached to the class diagram as operation descriptions. An example is followed through to illustrate these mechanisms. We have a practical implementation integrated to a real world UML modeling tool, the Nokia TED. This particular transform operation is an integral part of a larger framework of model operations used for checking, merging, slicing and synthesis of UML models.

1. Introduction

UML [2,13,18,19] has become an industrial standard for the presentation of various design artifacts in object-oriented software development. UML provides several diagram types, which view a system from different perspectives or at different levels of abstraction. Therefore, UML models of the same system share information and are dependent. Changes in one model may imply changes in another, and a large portion of one model may be synthesized on the basis of another.

Sequence diagrams provide a natural and easy medium for designing the desired behavior of the system. Examples of typical dynamic interactions of objects can be constructed at early phases of the software development process, often as refined representations of use cases [18,19]. After modeling examples of interactions, the designer should add the information implied by the sequence diagrams to the static model (class diagrams), or check that the static model conforms to the sequence diagrams. To achieve complete models, several iterations are typically needed: the models are gradually refined from rough sketches to specifications. Class diagrams are often annotated with pseudocode descriptions of operation implementations, attached as UML comments to class symbols. Such descriptions are frequently used, for example, in design pattern documentations [3].

Automated support for synthesizing one UML model from another can provide significant help for the designer. Such synthesis operations help the designer to keep the models consistent, to speed up the design process, and to decrease the risk of errors. Typically, every iteration phase increases the information present in the design. The new information must be somehow translated and merged to existing models. Usually this is done manually, which is often tedious and can easily lead to mismatches and inconsistencies due to human errors. Model synthesis is also an efficient way of creating intermediate documentation and can thus be used for enhancing communication between designers and management [14].

In this paper we discuss a particular UML transformation operation, which synthesizes a class diagram from a set of sequence diagrams. In addition, operation descriptions are synthesized from the same set of sequence diagrams and visualized as state machines. The state machines, in turn, are used as an input for a pseudocode generator, which outputs a sketch of the operation body in a textual format. Finally, the

generated class diagram can be annotated with the pseudocode presentations: a UML note that contains the pseudocode is attached to the corresponding operation. The mechanisms presented in this paper are primarily to be used during relatively early phases of software construction (analysis and design phases). We are not aiming at producing complete and consistent models or executable code, and we do not require this from the user either. We believe that a relaxed, pragmatic approach for modeling can benefit designers at early and intermediate phases of software development.

Tools for producing executable program code from design models are available. For example, with Rhapsody [7] and STATEMATE [6] the user can generate code from statecharts [5]. To be able to generate code that compiles and executes, the underlying statechart has to be exact, correct, and consistent. This, however, is not our goal. Instead, we provide automated support for constructing overviews of the current state of the design, generated from sequence diagram models that can be incomplete and imperfect. We do not require complete models.

Our implementation platform, the Nokia TED [21], is a multi-user software development environment that has been implemented at the Nokia Research Center. TED supports most of the UML diagram types and a reasonably large subset of the UML metamodel as a component library, thus allowing us to operate easily directly with the tool data. The transformation algorithms presented in this paper are implemented as COM components, interoperable with TED.

The paper is organized as follows. In Section 2 we give an overall description of the transformation process including the transformation operation from sequence diagrams to class diagrams, state machine synthesis, and pseudocode generation. An example of the transformation process is given in Section 3. Finally, discussion is presented in Section 4.

2. The transformation process

The synthesis of an annotated class diagram consists of three steps: class diagram synthesis, operation description synthesis, and pseudocode synthesis. These operations are implemented and integrated into TED.

A sequence diagram is transformed to a class diagram inside the standard UML metamodel [19]. We divide this transformation operation into two phases. First, we translate elements of a sequence diagram, such as classifier roles and messages, to corresponding elements of a class diagram, such as classes, associations, and methods. In addition to the fairly straightforward transformation, we define a set of heuristics that suggests more elaborate constructs, like composition, interface hierarchies, and multiplicities, to the user. These suggestions are uncertain by nature, but still plausible. By applying them we try to use the dynamic information and call patterns contained by a sequence diagram in order to generate additional structural information that might otherwise be lost during the transformation. The heuristics may be seen as "educated guesses" made by the system. The transformation process is defined in greater detail in [14].

Koskimies and Mäkinen have demonstrated how a minimal state machine can be synthesized from trace diagrams automatically [10,11,17]. This algorithm has been integrated with TED for synthesizing a state machine for a selected participating object from a set of sequence diagrams. The algorithm first extracts a trace from the sequence diagrams by traversing the lifeline of the object from the operation call down to the corresponding return message in each sequence diagram. The algorithm then maps items in the message trace to transitions and states in a state machine. Sent messages are regarded as primitive actions, which are associated with states. Each received message is mapped to a transition. In this research, the state machine synthesis algorithm is used as an intermediate step in pseudocode generation. To enable the generation of descriptive and understandable pseudocode from a state machine, we have modified the original synthesis algorithm to identify receivers for each method call.

The algorithm for generating pseudocode takes a state machine, interpreted as a deterministic finite automaton (DFA), as its input and generates, when possible, a pseudocode procedure represented by the particular DFA. There can be one or more statements associated with each action and a condition associated with each state transition. If there are several transitions between any two states, these transitions are combined together with a conjunction of the original transition labels. The language of the produced pseudocode consists of lexical symbols `begin`, `end`, `if`, `then`, `else`, `while`, `do` and `return`. The control

constructs of the language are *if-statements*, *while-loops*, and *do-while-loops*. We parse the DFA using five basic transformation rules and define a tail-recursive algorithm that parses the given DFA, trying to identify whether one of the rules applies to the current state. These rules define patterns in a DFA, which imply corresponding language constructs. The algorithm is described in more detail in [15].

It is obvious that only a subset of DFAs conform to the language defined by these simple rules. For example, interleaved loop and control structures cannot be expressed by these rules. It is possible to generate pseudocode from an arbitrary DFA using switch-case structures or goto-instructions. However, within our context we do not consider that kind of approach useful since the purpose of operation comments in UML is to given an intuitive overview rather than a detailed specification.

3. An example

As an example, consider a simple graphical user interface (GUI) dialog used for opening a personal file. The dialog consists of OK and Cancel buttons, and a list of names. This example resembles a typical design of a system involving user interactions.

In the simplified dialog example, we have an external user that interacts with a set of objects belonging to the GUI. When opening the dialog, the dialog object creates two button objects and a list object, reads status messages, and waits for the user to make a selection. After the user has selected a name and pressed OK, the dialog object reads the user input and dispatches an "openFile"- message to a folder object. Finally, some cleanup is done involving destruction of user interface objects. A sequence diagram describing successful execution is shown in Figure 1.

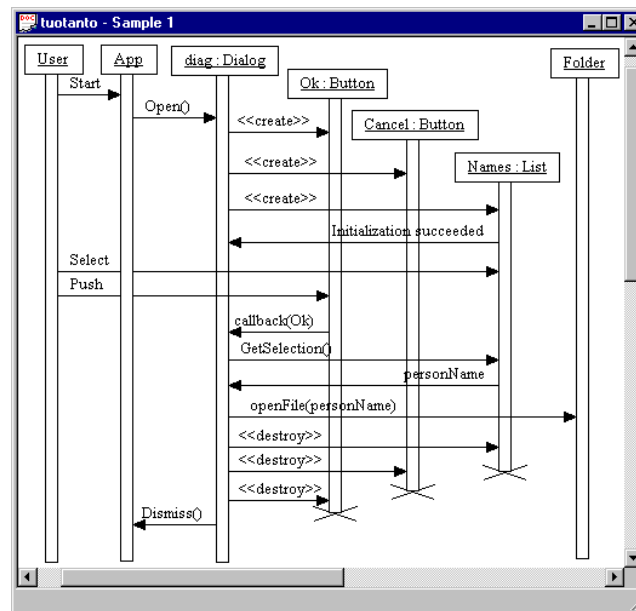


Figure 1. Example Sequence Diagram

In order to give a more comprehensive example, we construct two other sequence diagrams describing exceptional execution. The first one describes a case where the user presses the OK-button before making a valid selection, and the other one describes a situation where a dialog window cannot be correctly opened. We then call the state machine synthesizer component, which reads these three sequence diagrams from the TED repository, and synthesizes a state machine describing the execution of the "Dialog::Open()" operation. The view of the resulting state machine is depicted in Figure 2.

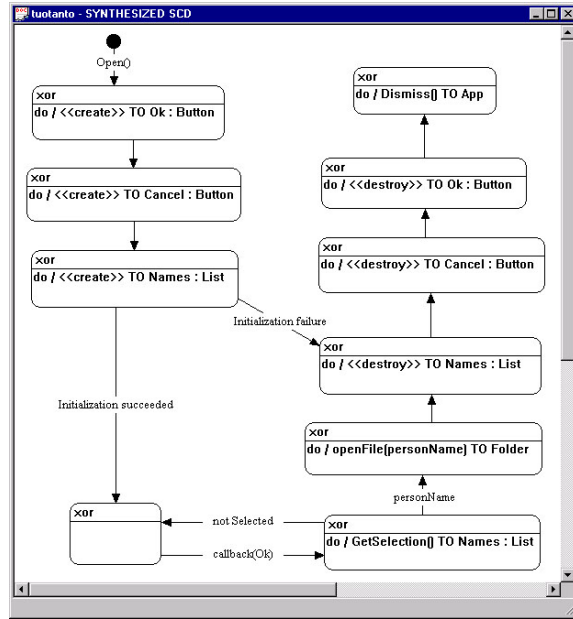


Figure 2. Synthesized State Machine

Figure 3 shows the annotated class diagram for the dialog example. In addition to the simple transformations (classes, associations, and operations), it also shows constructs generated by heuristics rules. We have two composition relationships, one from Dialog to List and one from Dialog to Button. These composition relationships derive from the fact that the objects of the two classes were created, destroyed and their operations accessed only by the dialog object. Furthermore, the broadcast heuristic applied has generated a many-multiplicity between Dialog and Button classes. This is shown as an asterisk (*) attached to the corresponding association. The pseudocode, generated from the state machine in Figure 2, shows a set object creation actions, a do-while loop with an enclosing if-structure and finally a set of object destruction actions. In short, the dialog object creates buttons, and if the initialization succeeded, wait for the user to make a selection. After a valid selection, it dispatches an “openFile” message to Folder.

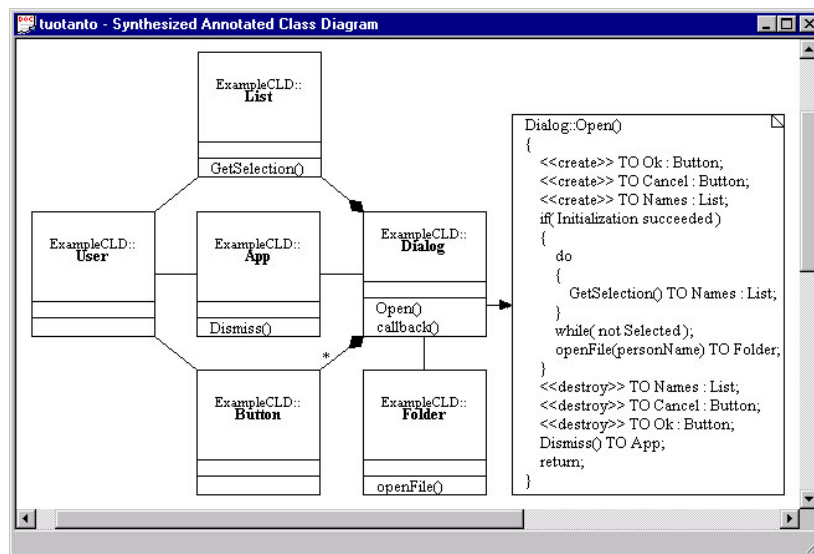


Figure 3. Synthesized Annotated Class Diagram

4. Discussion

In this paper we discussed a technique for synthesizing UML class diagrams from sequence diagrams. A synthesized class diagram can be annotated with operation descriptions, which are also synthesized from the same sequence diagrams. The operation descriptions are presented as pseudocode. The pseudocode generation consists of two steps: synthesizing state machines for operations and generating pseudocode from the state machines. The proposed techniques are implemented in a real-life UML design tool, the Nokia TED. We presented a step-by-step example of how the techniques were used in diagram synthesis. We also presented a number of possible extensions and future research topics.

The proposed techniques provide help for the designer in an early phase of the design process. The synthesized class diagram and operation descriptions do not aim to specify the system to be designed comprehensively; they are constructed from the incomplete information given as sequence diagrams and thus reflect the current state of the design. It is our belief that as such, the techniques presented in this paper can support the designer during software development by introducing an automated mechanism for translating model elements of the dynamic view to elements of the static view. The resulting synthesized class diagram can help a designer to gain a better insight of a system to be built, to communicate with other designers, and to compare and validate his or hers design against existing system models.

References

1. Biermann A.W. and Krishnaswamy, R. Constructing programs from example computations, *IEEE Trans. Softw. Eng.*, 2(3), 1976, 141-153.
2. Booch G., Rumbaugh J., Jacobson I. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.
3. Gamma E., Helm R., Johnson R., Vlissides J. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley 1994.
4. Godin, R., Mili, H. Building and Maintaining Analysis-Level Class Hierarchies Using Galois Lattices, in *Proc. of OOPSLA'93*, ACM Press, 394-410.
5. Harel D. Statecharts: A Visual Formalism for Complex Systems, *Science of Computer Programming*, 8, 1987, 231-274.
6. Harel D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., Shtull-Tauring, A., and Trakhtenbrot, M.. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *IEEE Trans. Softw. Eng.*, 16(4), 1990, 403-414.
7. I-Logix. On-line at <http://www.ilogix.com>, 2000.
8. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), ITU-T, Geneva, 1996.
9. Khriess I., Elkoutbi M., and Keller R. Automating the Synthesis of UML Statechart Diagrams from Multiple Collaboration Diagrams, in *UML'98: Beyond the Notation*, LNCS 1618, 1999, 132-147.
10. Koskimies K. and Mäkinen E. Automatic Synthesis of State Machines from Trace Diagrams. *Softw. Pract. & Exper.* 24(7), 1994, 643-658.
11. Koskimies K., Männistö T., Systä T., and Tuomi J. Automated Support for Modeling OO Software, *IEEE Software*, 15(1), January/February, 1998, 87-94.
12. Nørmark K. Synthesis of Program Outlines from Scenarios in DYNAMO. Aalborg University, 1998. On-line at <http://www.cs.auc.dk/~nørmark/dynamo.html>.
13. Rumbaugh J., Jacobson I., Booch G. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
14. Selonen P., Koskimies K., Sakkinen M. How to Make Apples from Oranges in UML. In *Proc. of HICSS-34*, 2000. To appear.

15. Selonen P., Systä T. Synthesizing Annotated Class Diagrams in UML. 2000. Submitted.
16. Schönberger S., Keller R., and Khriess I. Algorithmic Support for Model Transformation in Object-Oriented Software Development, in Theory and Practice of Object Systems (TAPOS), John Wiley & Sons, 2000.
17. Systä T. Static and Dynamic Reverse Engineering Techniques for Java Software Systems. Dept. of Computer and Information Sciences, University of Tampere, Report A-2000-4, PhD Dissertation, 2000.
18. The Unified Modeling Language Notation Guide v1.3. On-line at <http://www.rational.com>, 1999.
19. The Unified Modeling Language Semantics v1.3. On-line at <http://www.rational.com>, 1999.
20. Whittle J. and Schumann J. Generating Statechart Designs From Scenarios, in Proc. of ICSE'00, Limerick, Ireland, 2000, 314-323.
21. Wikman J. Evolution of a Distributed Repository-Based Architecture, On-line at <http://www.ide.hk-r.se/~bosch/NOSA98/JohanWikman.pdf>, 1998.