Dynamic Modeling in Forward and Reverse Engineering of Object-Oriented Software Systems

Tarja Systä
Department of Computer Science
University of Tampere
P.O. Box 607, FIN-33101 Tampere, Finland
cstasy@cs.uta.fi

Abstract

A prototype tool called SCED is used for modeling the dynamic behavior of object-oriented software as scenario diagrams and state diagrams. In SCED state diagrams can be synthesized automatically from scenario diagrams. When reverse engineering existing software, a parser and a debugger are used for extracting static and dynamic information, respectively. The parsed information is viewed as a nested graph using a reverse engineering environment Rigi. The debugged information is shown as SCED scenario diagrams and state diagrams. Static and dynamic views to the software can be improved and insured by comparing partly overlapping information generated by the parser and the debugger.

1. Introduction

Object-Oriented Analysis and Design (OOAD) methodologies support the designer in designing, visualizing, and documenting artifacts in object-oriented software systems. These methodologies provide notations and guidance to model both the static structure of the program and the dynamic behavior of the objects.

Variations of scenario diagrams and finite state machines are used in several OOAD methodoligies for dynamic modeling. In The Unified Modeling Technique (UML) [1] the corresponding diagrams are called sequence diagrams and statechart diagrams, respectively. In The Object Modeling Technique (OMT) [2] they are called event trace diagrams and state diagrams. A scenario diagram shows an object interaction arranged in time sequence during a particular execution of the system. Participating objects or classes are drawn as vertical lines and the interaction between them with horizontal arcs. A scenario diagram can also have participants outside of the system border, for example, a user giving inputs to the system. A state diagram shows the sequences of states that an object or an interaction goes through during its life in response to received stimuli, together with its responses and actions.

SCED [3] is a prototype environment built to support the dynamic modeling of object-oriented applications. SCED uses the OMT methodology as a guideline, although the resulting system could be useful for other methods as well, particularly for methods with a scenario driven approach. Despite the different purposes of scenario diagrams and state diagrams, they share common information. Hence, constructing one from the other can be partly automated. One of the basic observations behind SCED is that constructing scenario diagrams and fusing them into a state diagram can be supported by automated tools far more than is currently practiced. In [4], it has been demonstrated how a minimal state machine, which is able to execute all the given scenarios with respect to a certain object, can be synthesized automatically. This algorithm with few modifications has been implemented in SCED. On the other hand, scenario diagrams can be generated by animating the interaction of objects through a set of collaborating state diagrams. However, in contrast to conventional animation systems, in SCED one can add new behavior to the system during the animation process. This technique could be characterized as *design-by-animation*. By using state diagram synthesis and design-by-animation techniques in turns, the dynamic model can be refined semi-automatically: each iteration gives a more comprehensive set of scenario diagrams and more complete state diagrams.

Several tools are available for reverse engineering the dynamic behavior and static structure of existing software. Tools that focus on static aspects of the target system usually use parsers for extracting the software artifacts and their dependencies. Rigi [5] is an extensible and tailorable reverse engineering environment. The parsing system of Rigi supports several programming languages, and new parsers can

easily be added to it. The parsed information can be viewed as a directed graph using Rigi editor. Rigi also supports program slicing and building abstractions for the static views. These features are used for increasing the understandability and readability of the views.

The dynamic behavior of software can be extracted, e.g., by using a debugger, a profiler, or instrumenting the source code. Typical behavioral aspects to be extracted are: object and thread interaction, exceptions and errors, garbage collection, memory leaks, etc. A scenario is a natural, descriptive, and powerful way to record the object interaction. However, scenarios tend to grow rapidly when the target system gets more complicated. One way to deal with the scenario explosion is to detect behavioral patterns from the event trace. The automatic state diagram generation property of SCED provides another efficient way to deal with large event traces, to view the total behavior of a class of interest in a single view, and to examine its run-time behavior separately from the rest of the system.

2. Forward engineering

Most user interaction with SCED involves two independent editors: a scenario diagram editor and a state diagram editor. At any time while editing the scenario diagram, the user can select one participating object and synthesize a state diagram automatically for this object using a single menu command. The synthesis can be done for one scenario diagram only or for a specified set of scenario diagrams. Moreover, scenario diagrams can be synthesized into an existing state diagram.

When synthesizing a state diagram for an object, each scenario diagram is traversed from top to bottom from the point of view of a participant corresponding to that object. Each received event is mapped to a transition in the state diagram. Sent events are regarded as primitive actions that are associated with states. The synthesis algorithm attaches states to all actions and places at most one action into a single state. This is a restriction when OMT state diagram notation is considered. Hence, SCED provides algorithms for generating OMT state diagram notation for a synthesized and/or edited state diagram to simplify the state diagram while preserving its information. The generated OMT state diagram allows several actions placed into a state, actions attached to transitions, entry and exit actions of states, etc.

SCED scenario diagram notation differs slightly from UML or OMT ones. Some new concepts have been added in order to make SCED scenario notation more expressive. Like subroutines, a scenario diagram may consist of parts that have their own aims and characterizations. Such parts can be placed into a separate *subscenario* in SCED. These subscenarios can then be "called" instead of repeating their contents. SCED scenario notation also lacks some UML sequence diagram concepts, and some of the concepts have different graphical representations. For example, focus-of-control regions and timing constraints are not included in SCED scenario notation. The extended scenario diagram notation of SCED does not cause any major changes to the synthesis algorithm.

While the state diagram synthesis technique uses a set of scenario diagrams for generating a state diagram, design-by-animation technique uses a set of state diagrams for generating a scenario diagram. The state diagrams can simulate system behavior, sending events to each other and changing states according to received events. As long as external stimuli is not needed and the state diagram set represents a complete system, the event trace can be automatically generated. If that is not the case, the event tracing process halts whenever such undefined events are needed. In these cases the user helps the event tracing process to go on by providing the unknown behavior. Hence the resulting scenario diagram contains both automatically generated events and user defined events.

By using the state diagram synthesis and design-by-animation techniques in turns, a powerful design tool can be achieved. The dynamic modeling process is smoothly changed from a "water fall" type of modeling (first scenario diagrams, then state diagrams) to a more spiral and incremental way of modeling; the dynamic models for objects can be constructed semi-automatically by refining the state diagrams using a growing set of scenario diagrams and extending the scenario diagram set based on communicating state diagrams. Each iteration hence gives a more comprehensive set of scenario diagrams and a more complete state diagrams for the objects to be modeled. Each iteration also increases the degree of automation. The method is especially suited for modeling the behavior of one new component using the known behavior of other, predefined, and presumably correctly implemented components. For example, such predefined components could be classes belonging to a graphical user interface framework.

As a counterbalance to the state diagram synthesis property, scenario diagrams can also be desynthesized from the state diagram: the state diagram is updated by removing parts that correspond only to the scenario diagram to be desynthesized. Some support for consistency checking between scenario and state diagrams is available as well.

3. Reverse engineering

For fully understanding existing software both static and dynamic information need to be extracted. Static information includes typically software artifacts and their relations. In Java, for example, such artifacts could be classes, interfaces, methods, variables etc. The relations might include extending relationships between classes or interfaces,

method calls between methods, containment relationships between classes and methods or variables etc. Dynamic information contains software artifacts as well. In addition, it contains sequential information, information about concurrency and code coverage, etc.

Reverse engineering is not only applied to old legacy systems, it always needs to be part of forward engineering as well. In software development, reverse engineering the current static structure of the software helps the engineer to insure that the architectural guidelines are followed, to get an overall picture of the software, to document the implementation steps and so on. Reverse engineering the run-time behavior during the software development phase is essential. If the system seems to be irregularly unstable, tracing the bugs is possible only if the history and order of occured events is known.

The extracted information is not useful unless it can be shown in a readable and descriptive way. There are basically three kinds of views that can be used: static views, dynamic views, and merged views. The extracted information often consists of a large amount of detailed and low level software artifacts. Hence good views for showing the information is not usually enough, abstractions need to be build for making the views more clear and understandable. Figure 1 shows the main aspects of the problem.

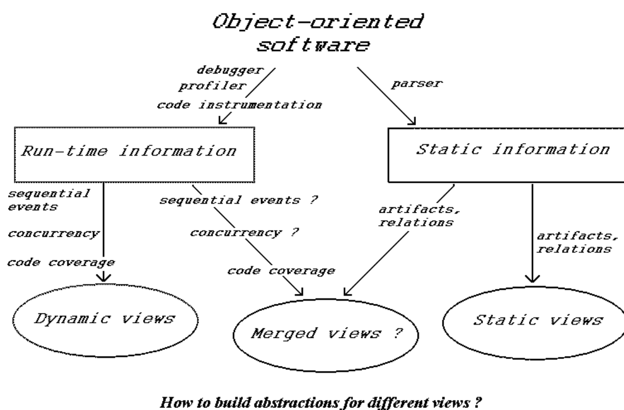

Figure 1. Different aspects of reverse engineering object-oriented software.

Rigi views static information in a form of a (nested) graph. It offers a graph editor and provides an extensible set of layout algorithms and algorithms used for program slicing and analyzing the software. Nested graphs enable showing the static structure of the whole system in a single view, and provide flexibility in browsing between different levels of abstractions built for the graph. This is an advantage compared to traditionally used class diagrams.

The state diagram synthesis facility of SCED provides an efficient way to view the total run-time behavior of one participant in a single view. The resulting state diagrams can then be used in design-by-animation approach when extending the design of the target system or designing another system that partly uses same classes as the current system.

When both static and dynamic information is extracted, also merged views can be used. Such views are usually formed by extending the static view by adding dynamic information into it. For example, code coverage information can be shown against a static view by giving weights for the cornensponding parts in the static view according their runtime usage. Merging static and run-time information has several advantages. First, the quality of the view can be insured by combining static and dynamic information. If both the parser and the debugger produces same source code artifacts and relations, the engineer can be quite confident that the artifacts are the right ones and the parser and the debugger works correctly. Second, the differencies between static and dynamic artifacts can be used to improve the views. For example, the parser cannot generate all default constructors if they are not explicitely written in the source code. The debugger can provide this piece of information. Third, merging information provides extended ways to do program slicing. For instance, based on dynamic information parts that are not used at run-time can be filtered out from the view. Slicing can also be made according to example scenarios.

Building abstractions for the views can be partly but not fully automated. Object-oriented languages support encaptulation. Such language structures can be used to build static abstractions automatically. For example, examing Java software by observing classes and their relations might clear the overall structure of the software, compared to studing it at the level of class members. In Rigi such abstractions can be built by collapsing all class method and variable nodes into a single class node, hence making the graph considerably smaller. Examing the structure in the class level might still contain too detailed information. The next step could be collapsing all classes and interfaces into packages, etc. Object-oriented metrics can also be applied for reasoning potential subsystems. Such subsystems could be groups of classes that are highly cohesive and have low coupling with other classes.

Dynamic abstractions typically differ from static ones. Building dynamic abstractions usually focuses on defining behavioral patterns and use cases. For example, initialization of a dialog might contain a sequence of events that are executed in a row every time the dialog is opened. Instead of repeated the whole event sequence, one single “dialog initialization” event could be considered. An example of a use case might be “withdrawing money using an ATM”. Such abstractions simplify sequence diagrams vertically: the number of events is decreased. Sequence diagrams can also be simpified horizontally by decreasing the

number of participants. This can be done by using the abstracted static model; sequence diagrams could show interaction between high level static compenents.

Building abstractions for merged views can be difficult, since the differencies between the static and the dynamic artifacts and their relations are not always complementary. For example, when overriding super class methods, polymorfism causes different method to be called than is actually written in the source code. Sequential information is often difficult to show in the same view with static information. In UML, collaboration diagrams can be used but the diagram gets difficult to read when the target software gets bigger. In general, the more information is added into a single view, the less readable it gets loosing its descritive power. Finally, building abstractions for merged views gets ambigious because dynamic and static abstractions usually differ considerably. When dynamic abstractions usually are behavioral patterns or use cases, static abstractions are subsystems. For example, most of the classes used by two use cases "withdrawing money using an ATM" and "paying a bill using an ATM" are the same and may belong to a single subsystem "ATM".

4. Current state of the research and future work

SCED is used for dynamic modeling in forward engineering of object-oriented software. The state diagram synthesis and design-by-animation features raise the level of automation in construction of the dynamic model. A prototype environment for reverse engineering Java software has been built. A Java source code parser is used for extracting static code artifacts and their relations for the target Java application or applet. The extracted information is viewed with Rigi editor. Rigi environment is also used for building abstractions and for program slicing. A Java source code debugger produces event traces consisting of basic object interactions. These event traces are shown as SCED scenario diagrams. The total behavior of an object can be viewed as a synthesized state diagram. Synthesized state diagrams can then be used in design-by-animation approach when designing new features for the target system. Some dynamic information is added to the static Rigi graph as well. The graph is extended with code coverage information and artifacts generated by the debugger but not recognizeded by the parser. Figure 2 shows the overall structure of the current system.

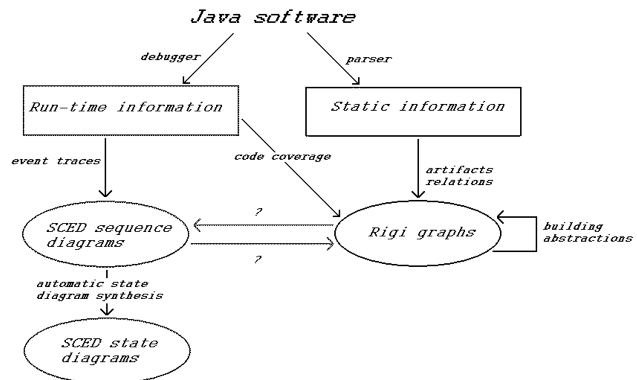

Figure 2. Current solution for reverse engineering Java software.

The emphasis in the future work is on examing how the dynamic and static views could contribute each other and when merged views could be used. Furthermore, the functionality of the Java debugger needs to be extended. Currently, the debugger produces method calls, constructor invocations, and thrown exceptions. However, the user should be given an option to choose information to be generated into the scenario diagrams. For example, in addition to currently generated events, the set of options might include: class variable assignments or accesses, if-else structures, repetition structures etc.

SCED has been built in a research project in co-operation with the University of Tampere, Tampere University of Technology, and several Finnish industrial partners. It is freely available at http://www.uta.fi/∼cstasy/scedpage.html or via ftp (ftp.cs.uta.fi, in directory /pub/sced). Rigi has been conducted by researchers in the Department of Computer Science at the University of Victoria. Rigi can be downloaded from http://www.rigi.csc.uvic.ca/.

I wish to thank Kai Koskimies and Hausi Müller for supervising my work. The SCED project has been financially supported by the Center for Technological Development in Finland (TEKES), the Nokia Research Center, Valmet Automation, Stonesoft, Kone, and Prosa Software. My current research is financially supported by Tampere Graduate School and Academy of Finland.

References

[1] Rational Software, *Unified Modeling Language, version 1.1*, [http://www.rational.com/uml/documentation.html], 1998.

[2] J. Rumbaugh et al, *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

[3] K. Koskimies, T. Männistö, T. Systä, and J. Tuomi, "Automated Support for Modeling OO Software", *IEEE Software*, **15**, 1, January/February 1998, pp. 87–94.

[4] K. Koskimies and E. Mäkinen, "Automatic Synthesis of State Machines from Trace Diagrams", *Software—Practice & Experience*, **24**, 7, pp. 643–658, 1994.

[5] H. MÏler, K. Wong, and S. Tilley, "Understanding software systems using reverse engineering technology", In *The 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences Proceedings (ACFAS)*, 1994.