Dynamic reverse engineering of Java software

Tarja Systä[1]

Abstract. An experimental environment has been built to reverse engineer the run-time behavior of Java software. Event trace information is generated as a result of running the target software under a debugger. The event trace is loaded to a prototype tool SCED [KMST98] as scenario diagrams. In SCED state diagrams can be generated automatically from a set of scenario diagrams. This facility is used to examine the total behavior of a desired class, object, or method as a state diagram.

1 Introduction

Software maintenance is complex and expensive because of program comprehension difficulties. Understanding the existing software is also important for re-engeineering purposes. Thus, the need for software engineering methods and tools that facilitate program understanding is compelling. A variety of *reverse engineering* tools provide means to help program understanding. Reverse engineering aims producing design models from the software. To fully understand existing software both static and dynamic information need to be extracted. Static information describes the structure of the software the way it is written in the source code, while dynamic information describes its run-time behavior. Both static and dynamic analysis yield information about the software artifacts and their relations. The dynamic analysis also produces sequential information, information about concurrency, code coverage, memory management and leaks, etc.

The adaption of object-oriented programming paradigm has changed programming styles dramatically. Extracting information about the dynamic behavior of the software is especially important when examining object-oriented software. This is due to the dynamic nature of object-oriented programs: object creation, object deletion/garbage collection, and dynamic binding make it very difficult, and in many cases impossible, to understand the behavior by just examining the source code.

In this paper we discuss an experimental environment that has been built to reverse engineer Java software. The static information is extracted from the byte code and analyzed with Rigi reverse engineering environment[MWT94].The dynamic event trace information is generated automatically as a result of running the target system under a debugger. SCED is used to view the event trace as scenario diagrams. In SCED state diagrams can be synthesized automatically from scenario diagrams. This facility is used

[1] University of Tampere, Box 607, 33101 Tampere, Finland (email cstay@cs.uta.fi)

to examine the total behavior of a class, an object, or a single method, disconnected from the rest of the system. The focus in this paper is on the dynamic reverse engineering part.

2 Used approach and tools

The static information is extraced from Java byte code, and the dynamic event trace information is generated as a result of running the target system under a debugger. Hence, Java source code is not needed. The byte code extractor and the debugger have been implemented using public classes belonging to *sun.tools.java* and *sun.tools.debug* packages, included in jdk1.2 . The event trace information generated at run-time contains method calls, constructor invocations, and thrown exceptions. They can be sent and received by methods, constructors, or static blocks of classes. Control flow information can also be generated. It consists of a notification of an executed conditional structure and an acknowledgement whether the condition yields **true** or **false**. In Java such conditional structures are **if**, **for**, and **while** and **switch-case** statements. In the formed event trace receivers and senders of events and owners of control flow structures can be set to be either classes or objects. When loading the event trace information to SCED they are represented as *scenario participants*, and shown as vertical lines. The experimental environment can be used as such to reverse engineer Java applications or applets. However, if the control flow information is needed, the line numbers can be read from the byte code only if has been compiled with a debugging option.

In SCED scenario diagram notation the basic message sequence chart notation has been exteded with some new concepts to make the diagram more expressive. Such concepts include *action boxes*, *assertion boxes,* and *state boxes.* The interpretation of an action box in this experiment is a sent event received by the object itself. State boxes are used to identify a conditional statement that will be executed next, and assertion boxes are connected to the usage of state boxes. The information whether the execution of the conditional statement results true or false is shown in an assertion box generated after the state box. A scenario in Figure 2 contains, e.g., a state box *"test line 366"* and an assertion box *"cond at 366 taken"*. The exact line numbers are read from the byte code.

In [KoM93], 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. 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. Assertion boxes are treated like received events, and action boxes like sent events. The synthesis algorithm defines states by their actions, not by their names. Names can be given to states by using state boxes in the scenario diagram or by editing the synthesized state diagram.

The user can synthesize a state diagram automatically for a selected scenario participant using a single menu command. The state diagram can be synthesized from a specified set of scenario diagrams. The synthesis algorithm works incrementally: scenarios can be

synthesized into an existing state diagram. Moreover, the user can synthesize a state diagram for a single method by selecting a method call event. The scenario is again traversed from top to down but only between the method call event and a corresponding *return* event. See Reference [KoM93, Sys97] for more detailed discussion.

3 Example: reverse engineering FUJABA software

For testing the usability of the approach explained in the previous section a target Java software has been examined. The selected target system FUJABA[RoH98] is freely available software developed in the University of Paderborn, Germany. The primary topic of the FUJABA project and environment is Round Trip Engineering with UML, SDM (Story Driven Modeling), Java and Design Patterns. FUBAJA is written in Java, containing almost 700 classes. The FUJABA version under examination was 0.6.3-0.

Next we focus on studing the functionality of the class diagram editor of FUJABA, and especially, a dialog used for defining and editing parameters of methods. The used dialog is shown in Figure 1. In this section we examine the behavior of the *modify* button and the behavior of the dialog itself. The class files of FUJABA are first parsed using a prototype byte code extractor. The resulting dependency graph consists of 25854 software artifacts (classes, interfaces, class/interface members etc.) and 109513 different kinds of dependencies between them. The graph can be analyzed using Rigi.

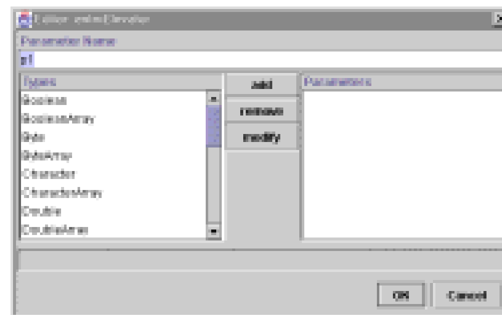

Figure 1. A dialog used for defining and editing parameters for a method.

Method *modifyButton_actionPerformed(java.awt.event.ActionEvent)* of the dialog class *de.uni_paderborn.fujaba.gui.PEParameters* is called each time the *modify* button is pressed. To capture the behavior of that method, breakpoints are set at the first line of all the methods, constructors, and static blocks that depend on it. The set of such methods can be found easily by running few simple scripts on the Rigi graph. In addition, to capture branching in the execution breakpoints are set for all conditional statements in the method *modifyButton_actionPerformed(java.awt.event.ActionEvent).*

The dialog in Figure 1 was used in a following way:

- The name of the selected parameter was changed and the *modify* button was pressed,
- The *modify* button was pressed when none of the parameters were selected,
- The type of a selected parameter was changed and the *modify* button was pressed, and
- The name of the selected parameter was deleted and the *modify* button was pressed.

Figure 2 shows the scenario resulting from the first case, and 3 shows the total behavior the method *modifyButton_actionPerformed(java.awt.event.ActionEvent)* of class *de.uni_paderborn.fujaba.gui.PEParameters*, synthesized from four scenarios.

Figure 2. A scenario generated when a parameter was edited using the dialog in Figure 1.

Figure 3. The behavior of *modifyButton_actionPerformed(java.awt.event.ActionEvent)* method of class *de.uni_paderborn.fujaba.gui.PEParameters*. The state diagram describes the control flow of the method.

The total behavior of the dialog in Figure 1 is examined next. The breakpoints are again set at the first line of all methods, constructors, and static blocks that depend on any of the methods of the dialog class *de.uni_paderborn.fujaba.gui.PEParameters.*
The dialog was used in a following way:

- The dialog was opened, a parameter name and type was defined, the *add* button was pressed, and the *ok* button was pressed
- The dialog was opened, a name of the parameter was changed, the *modify* button was pressed, and finally the *ok* button was pressed
- The dialog, was opened, the *remove* button was pressed (a parameter was selected), and the *ok* button was pressed.

Note that the above usage of the dialog does not cover all possible ways the dialog can be used and hence cannot define the total behavior of the system. Alltogether 12 scenarios result when using the dialog in the described way. The state diagram synthesized for the dialog class *de.uni_paderborn.fujaba.gui.PEParameters* consists of 49 states. Figure 4 shows the right down corner of it. Assumingly, the more the dialog is used, the less the synthesized state diagram will grow: rather than more states, more transitions would be generated describing different paths through the state diagram. When using the dialog the *modify* button was pressed once after changing the name of a parameter. In case of modeling the bahavior of the button itself it was used more and in several different ways. If that usage of the *modify* button was part of the usage when modeling the bahavior of the whole dialog, then the state diagram in Figure 3 would be a subdiagram of the state diagram generated for class *de.uni_paderborn.fujaba.gui.PEParameters*. Now, only part of the state diagram in Figure 3 can be found from it. This part is shown in Figure 4.

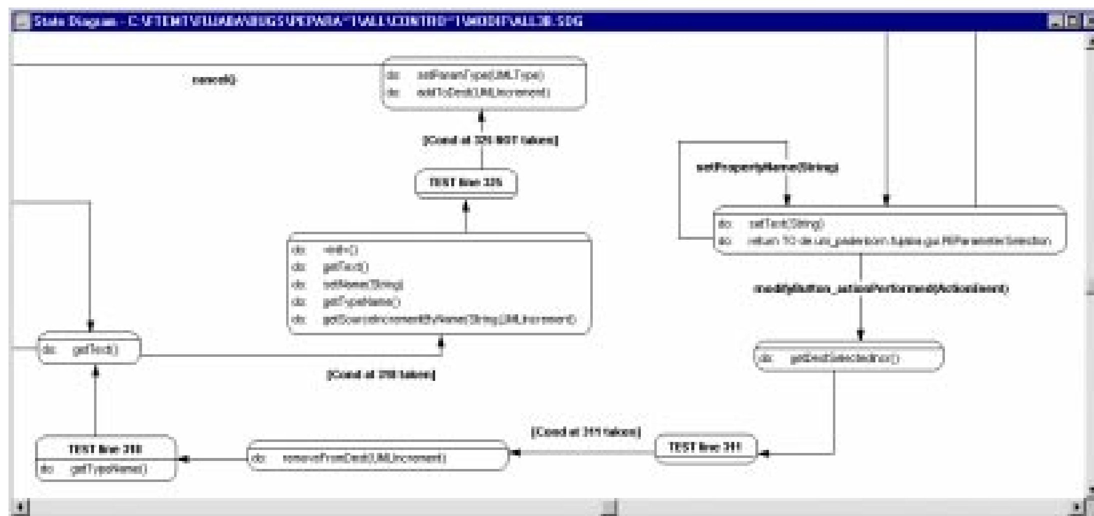

Figure 4. A right down corner of a state diagram generated for the dialog class *de.uni_paderborn.fujaba.gui.PEParameters*. The original state diagram contains 49 states.

When reverse engineering FUJABA software state diagrams were synthesized for selected methods, objects, and classes from several sets of generated scenario diagrams. Based on the feedback received from FUJABA group the resulting dynamic models were self-explanatory and they correspond closely to their understanding of the behavior. In addition, a source of one known bug was identified by examining the generated scenarios.

4 Discussion

The application of reverse engineering and re-engineering techniques are not limited to understanding and modifying old legacy systems. They can and should be applied for forward engineering purposes 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 for profiling, tracing the sources of bugs, understanding and insuring the current behavior of the software, finding dead code, etc. Similarly, re-engineering is always part of forward engineering. Applying reverse engineering techniques during the software development phase also supports documentation, hence avoiding a similar situation with our Java code that we currently have to phase with COBOL and C code.

Several tools are available to reverse engineer the static structure of object-oriented software as class diagrams, but there are not many tools that support reverse engineering standard dynamic models for it. However, it is desirable that the reverse engineering tool is able to produce similar diagrams as are used and needed in forward engineering. By combining the described approach to construct scenario and state diagrams for the software with some static reverse engineering tool, more support for re-engineering object-oriented application can be given.

References

[KMST98] Koskimies K., Männistö T., Systä T., and Tuomi J.: Automated Support for Modeling OO Software, *IEEE Software,* **15**, 1, Janyary/February 1998, pp. 87-94.

[MWT94] Muller H., Wong K., and Tilley S.: Understanding software systems using reverse engineering technology, In *The 62nd Congress of L'Association Cadadienne Francaise pour l'Avancement des Sciences Proceedings (ACFAS),* 1994.

[KoM93] Koskimies K. and Mäkinen E.: Automated Synthesis of State Machines from Trace Diagrams, *Software-Practice &* Experience, **24**, 7, pp. 643-658.

[Sys97] Systä T.: *Automated Support for Constructing OMT Scenarios and State Diagrams in SCED,* University of Tampere, Dept. of Computer Science, Report A-1997-8, 1997.

[RoH98] Rockel I. And Heimes F.: *FUJABA - Homepage,* [http://www.uni-paderborn.de/fachbereich/AG/schaefer/ag_dt/PG/Fujaba/fujaba.html], February, 1999.