Understanding the Behavior of Java Programs

Tarja Systä
Software Systems Laboratory
Tampere University of Technology
P.O. Box 553, FIN-33101 Tampere, Finland
tsysta@cs.tut.fi

Abstract

To fully understand the underlying architecture of an object-oriented software system, both static and dynamic analyses are needed. Dynamic reverse engineering techniques are especially important for understanding the run-time behavior of objects in a distributed object systems and in systems that rely heavily on polymorphism.

Shimba, a prototype reverse engineering environment, has been built to support understanding an existing Java software system. The dynamic event trace information is generated automatically as a result of running the target system under a customized sdk [14] debugger and viewed as scenario diagrams using the SCED tool [5]. In SCED, state diagrams can be synthesized automatically from scenario diagrams. This facility is used to visualize the total behavior of a selected object or method, disconnected from the rest of the system. This paper demonstrates how Shimba aids understanding the behavior of Java programs. A case study in made to validate the usefulness of the approach. The system under investigation is the FUJABA environment consisting of almost 700 Java classes.

1. Introduction

This paper discusses *dynamic reverse engineering* techniques for modeling the run-time behavior of Java software systems. An experimental environment called Shimba has been built to reverse engineer Java applications and applets. The static information can be extracted from Java byte code and viewed using the Rigi reverse engineering tool [8]. The dynamic event trace information is generated automatically as a result of running the target system under a customized sdk debugger. The debugger is implemented using public classes of *sun.tools.debug* package of sdk1.2. The event trace can be saved in SCED scenario diagram format and viewed with the SCED tool [5]. SCED is a dynamic modeling tool that provides two independent editors for *scenario diagrams* and *state diagrams*. Scenario diagrams and state diagrams are variations of UML sequence diagrams and statechart diagrams, respectively.

Many dynamic reverse engineering tools use variations of Message Sequence Charts (MSCs) to visualize the run-time behavior of the target object-oriented software system [2, 3, 6, 7, 10]. In Shimba, the visualization of the run-time behavior of object-oriented software has been taken one step further: not only scenario diagrams but also the final specification of the dynamic behavior, i.e. the state diagram, is composed automatically as a result of the execution of a target system. The step is made possible by using an automated state diagram synthesis feature of SCED. Generated state diagrams allow the user to examine the dynamic behavior from a different angle compared to scenario diagrams. While scenario diagrams show the interaction between several objects, a state diagram shows the total behavior of a certain object or a method in the system. Optionally, information about the dynamic control flow of selected objects can be extracted and added to the scenario diagrams. Using the state diagram synthesis feature of SCED, the dynamic control flow of a selected object or a method can be visualized as a state diagram.

Managing and abstracting the huge amount of event trace information included in MSCs is one of the most challenging problems for dynamic reverse engineering tools. Some of the tools support techniques to search behavioral patterns, which then can be used to manage the event trace information [3, 10, 9]. Shimba also supports this approach. It provides string matching algorithms for recognizing behavioral patterns in event traces. The found patterns can be visualized and analyzed with the SCED tool.

Shimba supports both static and dynamic reverse engineering of Java software systems. In [16], relationships between SCED scenario diagrams and Rigi views were used to better understand the target system using Shimba. Connections between static and dynamic views provide extended means to analyze and understand the target system. For example, run-time information can be reflected on the static model and scenario diagrams can be used to visualize the interactions among high-level static components. These techniques to combine static and dynamic analysis of the target software system are discussed in [16]. This paper, in turn, demonstrates how Shimba aids understanding the behavior of Java programs by further analyzing the generated scenario diagrams. The event trace information included in the scenario diagrams is used to build high-level dynamic views applying two techniques: state diagrams are generated for selected objects to view their overall behavior and string matching algorithms are applied to search behavioral patterns from the scenario diagrams. A case study, in which several dynamic reverse engineering tasks are set, is carried out to validate the usefulness of the approaches.

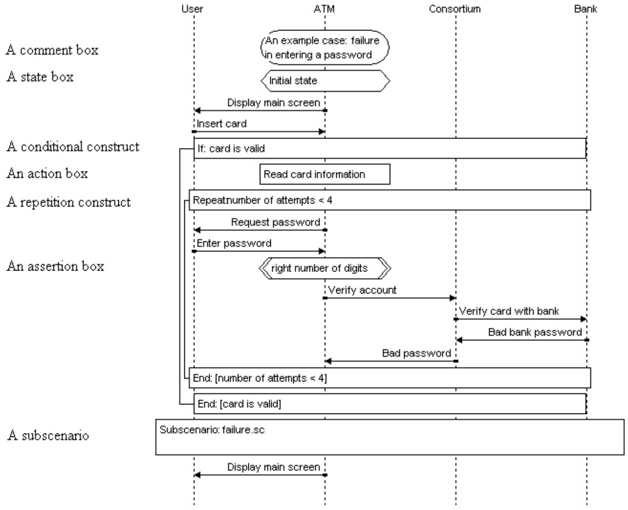

Figure 1. SCED scenario diagram notation concepts.

2. Dynamic modeling using SCED

SCED [5] is a prototype environment that has been built to support the dynamic modeling of object-oriented applications. Most user interaction with SCED involves two independent editors: a scenario diagram editor and a state diagram editor. In the SCED scenario diagram notation, the basic MSC notation has been extended with some new concepts to make the diagrams more expressive. Such concepts include *action boxes*, *assertion boxes*, *state boxes*, *conditional constructs*, *repetition constructs*, and *subscenarios*. When applied for dynamic reverse engineering purposes in Shimba, an action box is used to visualize a sent message that is received by the object itself. The usage of assertion boxes in Shimba is connected to the usage of state boxes. They are generated to indicate which branches of the conditional statements have been taken. The behavioral patterns in the event trace are visualized with repetition constructs and subscenarios. Semantically, a subscenario is a shorthand notation for importing the contents of the referred scenario diagram at the position of the subscenario box. Conditional and repetition constructs as well as subscenarios can be nested to an arbitrary depth. The SCED scenario diagram notation is shown in Figure 1. In general, concrete scenario diagram elements (following the SCED scenario diagram notation) are called *scenario items*.

There are semantically corresponding items in the UML sequence diagram notation [13, 11] for most of the SCED scenario diagram concepts. For example, assertions, conditional constructs, and repetition constructs correspond to constraints, conditional branching, and iteration, respectively. However, UML sequence diagram notation does not include a corresponding notational concept to a subdiagram box in SCED, i.e., a way to nest sequence diagrams. In addition, UML sequence diagrams are not practical for expressing a repetition of a set of sequent, disconnected messages. Repetition constructs and subscenarios thus make SCED scenario diagrams more applicable for modeling behavioral patterns than UML sequence diagrams. On the other hand, UML sequence diagram notation include concepts that can not be expressed with SCED. For example, distinguishing active and passive objects is not possible in SCED scenarios.

State diagrams notation in SCED can be characterized as a simplified UML statechart notation. The main restriction is that SCED state diagrams can not be used to express concurrency. Also, history states, submachine reference states, and timing constraints are not included in the SCED state diagram notation.

In [4], it has been demonstrated how a minimal state machine can be synthesized from trace diagrams automatically. This algorithm with few modifications has been implemented in SCED for synthesizing a state diagram for a selected object from a set of SCED scenario diagrams [15]. The algorithm first extracts a trace from the scenario diagrams by traversing the vertical line of the object from top to bottom in each scenario diagram. The algorithm then maps items in the event trace to transitions and states in a state diagram. Sent events are regarded as primitive actions that are associated with states. Each received event is mapped to a transition. The additional SCED scenario diagram concepts do not cause major changes to the basic algorithm. Assertion boxes are treated like received events, and action boxes like sent events. Algorithmic constructs (i.e., conditional and repetition constructs) and subscenarios are read recursively.

The synthesis algorithm attaches states to actions and places at most one action into a single state. This is a restriction when UML state diagram notation is considered. Thus, SCED provides algorithms for generating advanced state diagram notation for a synthesized and/or edited state diagram to simplify the state diagram while preserving its information [17]. The generated concepts are similar to those used in UML. The simplified state diagram allows, for instance, several actions placed into a single state, entry and exit actions of states, and actions attached to transitions.

The user can select one participant in a scenario and synthesize a state diagram automatically for it using a single menu command. The synthesis can be done for one scenario diagram only or for a specified set of scenario diagrams. The state diagram synthesis algorithm works incrementally: scenarios can be synthesized into an existing state diagram. Moreover, state diagrams can be synthesized for single methods. The synthesis algorithm is described in more detail in [4, 15, 17], and the state diagram simplification algorithms are explained in [15, 17].

3. Collecting information

To collect event trace information, the target Java system is executed under a customized sdk debugger. The user can select the classes and/or methods for which the event trace information is collected. The user can also choose exceptions to be tracked. Event trace information is generated when methods are called and exceptions are thrown. If the thrown exception is selected by the user, then the receiver for the event is set to be the class of the exception. Otherwise, the receiver will be the superclass of all exceptions, namely *java.lang.Exception*.

When the engineer is interested in the behavior of a specific part of the software, it is not meaningful to generate information about all the object interaction of the system, but only the interaction that has an effect on the behavior of the part of interest. Such information prefiltering can decrease the size of generated event traces dramatically while still containing the information of interest. In order to be able to do that the engineer needs to know which classes depend on each other, i.e., knowledge about the static structure of the software. In Shimba, information about the static software artifacts and their dependencies is extracted from the byte code Java class files. The following software artifacts are recognized: classes, interfaces, methods, constructors, variables, and static initialization blocks. The dependencies extracted among them include: an extension relationship (e.g., a class extends another class), a containment relationship (e.g., a class contains a method), a call relationship (e.g., a method calls another method), an access relationship (e.g., a method accesses a variable), an assignment relationship (e.g., a method assigns a value for a variable), and so on. The information can be viewed and examined using the Rigi reverse engineering environment [8]. In Rigi all the software artifacts are shown as nodes, and relationships as directed edges between two nodes. The type of a node or an edge is identified by a specific color. Information about parts of the software that depend on each other in one way or another can be easily generated using Rigi, even for an unknown system, by running a simple query (a script) on the static dependency graph. By filtering out the rest of the graph, the engineer can quickly pick up the classes and their member functions that depend on or are depended by the part of the software of interest.

To capture the event trace information, breakpoints are set for the debugger. Breakpoints are set at the first line of all methods, constructors, and static initialization blocks of the selected classes. The dynamic control flow information is also generated using breakpoints. The conditional statements to detect are **if**, **for**, **while**, **do-while**, and **switch-case** statements. The debugger keeps a stack of previously activated class members. The stack information can be used to conclude the sender of the event, generated when a breakpoint is hit. After a new SCED scenario event is created, the execution of the program continues automatically. A hit of a breakpoint causes only a short pause of the execution of the system, giving the user no influence on deciding about the continuation of the execution.

The conditional statements indicate the branching points in the control flow. To capture the dynamic control flow information, state boxes and assertion boxes are added to the SCED scenario diagrams corresponding to the condition to be evaluated and the resulting value (i.e., the branch that has been taken). For example, a state box *"TEST line 311"* in a SCED scenario diagram in Figure 4 indicates that in the source code there is a conditional statement at that line 311. An assertion box *"Cond at 311 taken"*, in turn, indicates that binary construction that corresponds to the condition in the source code yielded true. Note that Shimba is independent from the source code of the target software. Hence, state boxes are not labeled by the actual conditional structures, even though that would be more descriptive. If the source code existed, that would be possible since the line numbers are known (if the source code has been compiled using a debugging switch). There are three major reasons for chosen labels for state and assertion boxes. First, we want to be totally independent from the source code. Second, since we do not give the condition written in the source code, we at least want to provide the user a possibility to quickly find that line if she wants to. Third, it does not seem to be more useful nor more descriptive to add the actual byte code instructions that caused the insertion of the state and assertion boxes.

4. Managing the explosion of the event trace

A large amount of event trace information results from even a relatively brief usage of the target system. Thus, methods for managing the event explosion problem are needed. Applying pattern matching algorithms is one way to deal with the problem. This approach is used, for example, in ISVis [3] and Jinsight [2]. In Shimba, the original scenario diagrams can be modified by applying Boyer-Moore string matching algorithms [1] to them. The found patterns provide means to raise the level of abstraction of the scenario diagrams and to decrease their size.

When applying string matching algorithms to SCED scenario diagrams, the behavioral patterns found are shown either as repetition constructs or as subscenarios. Repetition constructs are used if the pattern is repeated at least twice in a row. Patterns that occur in a row are potentially generated by **while**, **for**, and **do-while** structures. Hence, a repetition construct is a natural way to show the same situation in the scenario diagram. The name of the repetition construct indicates the number of times the pattern has been repeated. Repeated events may appear also in other circumstances. For example, when debugging the behavior of a GUI software, moving a mouse, clicking a mouse, keyboard commands, etc., often cause an event to be repeated several times. In such situations a single event might be repeated a huge number of times, even when compared to the size of the event trace. Hence repetition constructs for repeated single events are generated already during the debugging process, i.e., when generating the original scenario diagrams.

Behavioral patterns may also occur separated from each other. For example, in a GUI software opening of a dialog box causes several events to be executed in a row (initialization of the dialog). Such a pattern is repeated every time a dialog is opened. These patterns may appear in different scenario diagrams, or at least, disconnected from each other. Subscenarios are used to visualize such patterns. Subscenario boxes provide a powerful way to link scenario diagrams: instead of repeating the event sequence in a place of their appearance, a subscenario box is used to refer to another scenario diagram that actually contains the event sequence. When double clicking a subscenario box, the scenario diagram containing the detailed pattern information is opened. Subscenarios help the engineer to recognize the patterns, browse them, and structure the described behavior. The synthesis algorithm reads the events of nested scenario diagrams recursively. The resulting state diagram is similar to one generated for the set of original, unnested scenario diagrams.

Another way to manage huge event traces is to split them into several shorter, more manageable ones. This approach is used in Shimba. The event trace is split automatically into several scenarios in order to limit the size of a single scenario. The user can also influence on saving the event trace as scenarios: at any point during the execution of the target system the user can dump the current event trace into a scenario file and initialize the event trace by choosing a single menu command. Hence, the user can "record" the desired interval of the execution.

5. A case study

To validate the usefulness of the approach explained in the previous section a target Java software system called FUJABA [12] was examined. The primary topic of the FUJABA project and environment is Round Trip Engineering with UML, SDM (Story Driven Modeling), Java and Design Patterns. FUJABA is written in Java, containing almost 700 classes. The FUJABA version under examination was 0.6.3-0.

The focus on this case study was on studying the functionality of the class diagram editor of FUJABA. Several dynamic reverse engineering tasks were set for the case study. Most of these tasks are goal-driven. We use Shimba to analyze the behavior of specific objects and methods and to seek answers to questions of the following form:

1. How does a certain part of the software behave?
2. What are the methods that have call dependencies with this part?
3. What is the overall run-time usage of the method or the object?
4. What is the dynamic control flow of the method or the object?
5. Was the run-time usage diversified enough to produce information that covers all possible use cases?

6. How can a certain state in the object's life be reached and how does the execution continue ?

7. To which messages has an object responded at a certain state during its lifetime?

8. Does the run-time behavior contain regular behavioral patterns that are repeated? If it does, what are the patterns and in which circumstance do they occur?

5.1. Modeling the internal behavior of a method

Figure 2 shows a dialog box used in FUJABA for defining and editing parameters of methods. In the case study, editing parameter of methods was especially closely examined, since there was a known bug in parts of FUJABA needed for that. The bug is reported in the FUJABA bug report [12] and has been fixed in the later versions of FUJABA. Next we focus on examining the internal behavior and the run-time control flow of the *modify* button. Each time the *modify* button is pressed, method *modifyButton_actionPerformed(ActionEvent)* of the dialog class *PEParameters* is called.

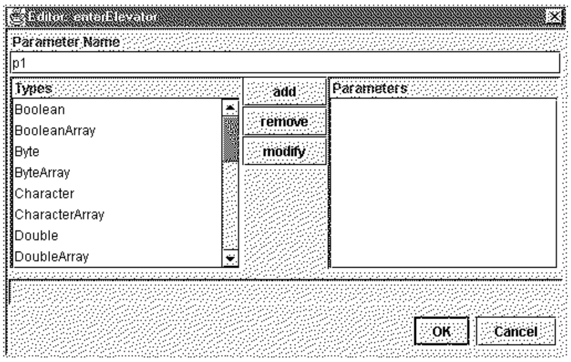

Figure 2. A dialog used by a class diagram editor of FUJABA.

By running few scripts in Rigi, the method *modifyButton_actionPerformed (ActionEvent)* of class *PEParameters*, nodes that depend on it, and nodes that are referenced by it can be easily separated from the rest of the system. This is depicted in Figure 3. From the dependencies 11 method nodes and 3 constructor nodes have been selected. All the other nodes have been filtered out. Also nodes representing software artifacts that do not belong to FUJABA itself (e.g., those belonging to sdk or Swing) have been filtered out. This has been done assuming that the behavior of such classes does not considerably help in understanding the behavior of the target system itself. For example, methods of the class *java.lang.String* are called frequently but such calls are meaningless for understanding

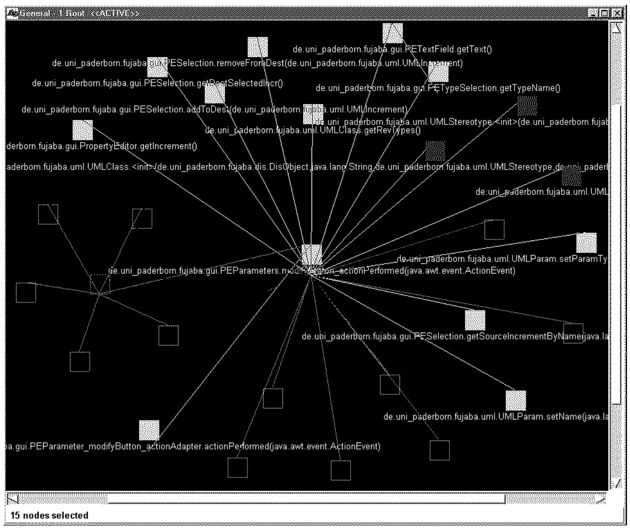

Figure 3. A Rigi graph that shows the method modifyButton_actionPerformed(ActionEvent) and its dependencies.

the behavior of the dialog in Figure 2. Furthermore, it can be easily certified (by running a couple more scripts) that none of the methods that are called by the method *modifyButton_actionPerformed (ActionEvent)* is overridden in any subclasses. Nodes that represent the overridden methods should naturally be included in the dependency graph.

The debugger set breakpoints automatically for the methods and constructors visualized in Figure 3. According to the user's instructions, breakpoints were set to capture both object interaction and dynamic control flow information. The dialog in Figure 2 was used four times in the 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 was selected, the type of the 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 4 shows the scenario diagram resulting from the first case, and Figure 5 shows the dynamic control flow of the method *modifyButton_actionPerformed(ActionEvent)* of class *PEParameters*, synthesized from four scenario diagrams. A state diagram is a powerful and natural graphical representation to examine the dynamic control flow and the internal behavior of the method. Dynamic control flows are useful for detecting decision making, for profiling, for investigating code usage, etc. Since the information is generated automatically based on the usage of the target software, the user can get only those pieces of information she is interested in and thus generate desired dynamic control flows.

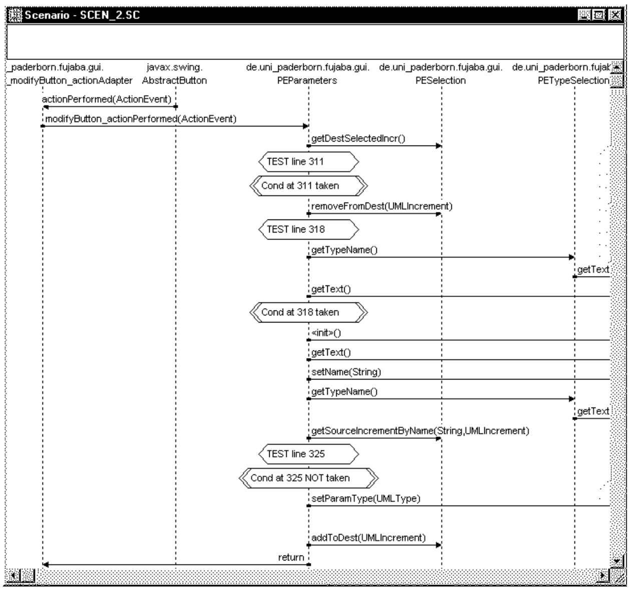

Figure 4. A scenario diagram describing one execution path through the method modifyButton_actionPerformed(ActionEvent).

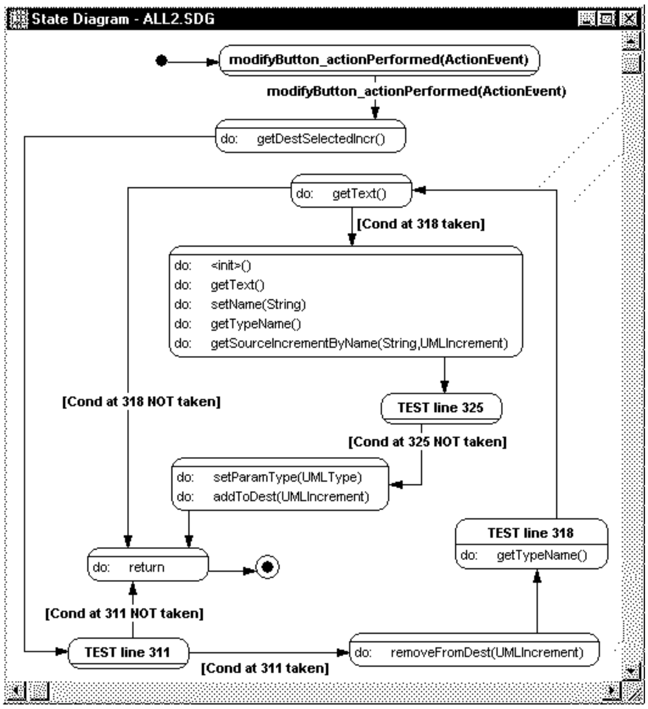

Figure 5. The behavior of the method modifyButton_actionPerformed(ActionEvent).

The synthesized state diagram can also be used to estimate if the "test cases" (usage of the software) are covering enough. By comparing it to the source code or the static model, it can also be used to estimate if the method in question includes unreachable code. The points in the state diagram that reveal such pieces of information are the states with state names. A complete state diagram should show all possible branches. In other words, there should be (at least) two different paths leading out from each state that identifies a branching point. For example, the state diagram in Figure 5 is not complete, because from the state *"TEST line 325"* there is a single path to the final state. Note that this can not be concluded from the state *"TEST line 318"*, since the actual branching point is the next state, that is, a state with the action *getText()*.

When examining the overall behavior of the dialog in Figure 2, Rigi was again used to identify all the nodes that belong to the dialog, nodes that depend on them, and nodes that are referenced by them. Breakpoints were set at the first line of all chosen member functions and for all conditional statements.

The dialog was used three different times while running FUJABA (i.e., three instances of the class *PEParameters* were created). The objects were not alive at the same time. The usage of the dialog resulted in altogether 20 scenario diagrams. The state diagram describing the dynamic control flow of the dialog contains 48 states. Such a large state diagram is difficult to be read and visualized. Although a relatively simple and brief usage of the system produces a large state diagram, it can be assumed that 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 behavior of the button itself it was used four times in different situations. If that usage of the *modify* button was part of the usage when modeling the behavior of the whole dialog, then the state diagram in Figure 5 would be a subdiagram of the state diagram generated for the class *PEParameters* itself.

When generating information about the object interaction only (ignoring the dynamic control flow information), a smaller and slightly more abstract state diagram diagram shown in Figure 6 resulted. The dialog was used exactly the same way as in the case discussed above.

A class is a basic software component in an object-

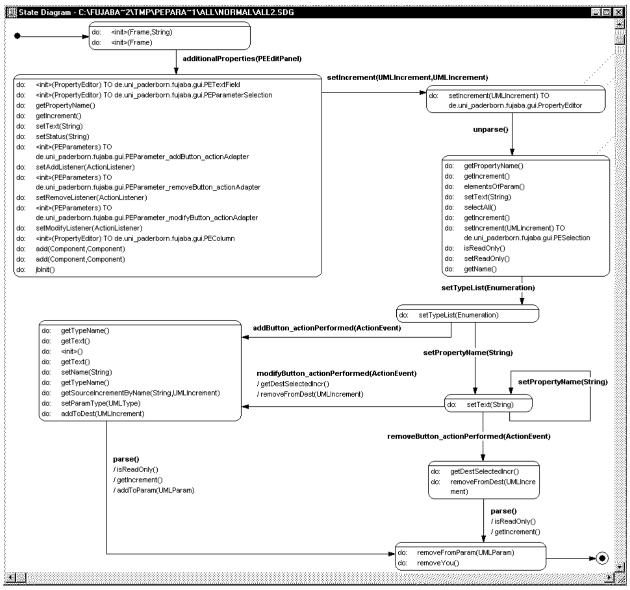

Figure 6. A state diagram showing the method invocations related to the dialog in Figure 2.

oriented software system. Understanding its usage and the behavior of its instances is important (e.g., for reverse engineering, reuse, and forward engineering purposes). Such pieces of information would thus be valuable to be added to the documentation of the software.

5.2. Structuring scenarios with behavioral patterns

In Shimba, SCED scenario diagrams provide a view and an editor to browse the exact sequential event trace information. Since, the amount of scenarios grows rapidly during the execution of the system, behavioral patterns are used for structuring, reorganizing, and packaging the scenario diagrams without changing their information contents. The patterns are searched using algorithms that apply the Boyer-Moore string matching algorithm [1]. Thus, the behavioral patterns are recognized based on strict string-based matches only. A pattern is a sequence of any SCED scenario diagram items repeated at least once.

In FUJABA, the dialog in Figure 2 is called by another dialog that is used to define and edit methods of a class. The class that implements this dialog is called *PEMethod*. To analyze the behavior of this dialog and thus studying further how methods are constructed using the class diagram editor of FUJABA, run-time information for the class *PEMethod* was generated. The application of the scenario structuring algorithms results in a regular hierarchy of a set of scenario diagrams. The algorithms were applied for six scenario diagrams that followed from a brief usage of FUJABA. During the FUJABA session, a dialog for editing methods of a class was used a few times. The event trace information was generated for the dialog class *PEMethod* itself, for its superclasses *gui.PETextEditor* and *PropertyEditor*, and for class *UMLMethod*. The superclasses were easily found using Rigi. Figure 7 shows one of the scenario diagrams after it has been modified by the scenario structuring algorithms. The algorithms generated two subdiagram boxes and one repetition block with label *"times: 4"*. The other repetition block that contains a single scenario item was included in the original scenario diagram. The scenario diagram depicts the object interaction that was involved when the dialog was opened for a class that has four methods. The dialog was exited right after opening it by pressing the *cancel* button.

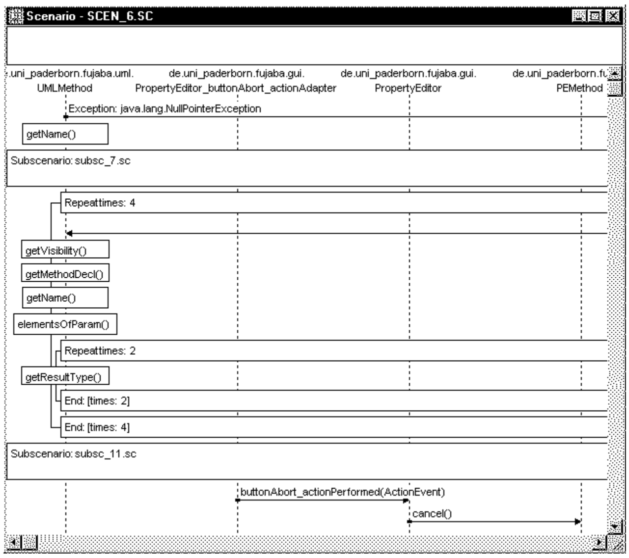

Figure 7. A scenario diagram after applying the scenario structuring algorithm for it.

The object interaction that was needed for the initialization of the dialog was repeated every time the dialog was opened. The scenario structuring algorithm recognized that pattern and formed the subscenario box *subsc_7.sc* for it. Figure 8 shows the contents of the subscenario box *subsc_7.sc*. The original scenario diagrams consist of 473 scenario items. After applying the scenario structuring algorithm the total number of scenario items was reduced to 201, giving 58% savings. The proportion of decreased scenario items naturally varies depending on the set of scenario diagrams. When same or similar parts of the target system are used several times during the execution session (as in this case), the proportion of saved scenario items is typically high.

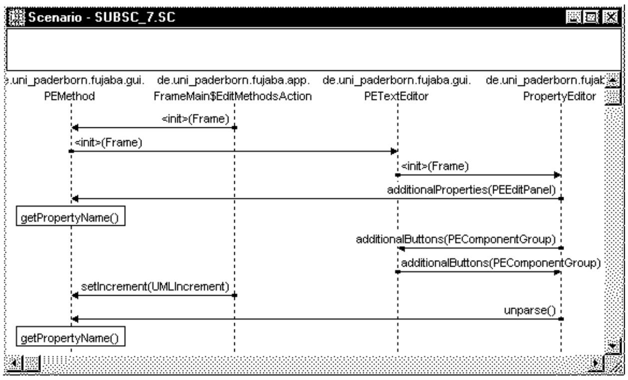

Figure 8. The contents of the subdiagram box subsc_7.sc in Figure 7.

5.3. Modeling the behavior of a thread

FUJABA uses several threads (i.e., subclasses of class *java.lang.Thread*) that are running concurrently. Most of them have a common superclass, namely *AEngine*. In Java, all classes whose instances are intended to be executed by a thread must implement the *run()* method of the *java.lang.Runnable* interface. The behavior of a thread is mostly defined in the *run()* method. The *run()* method is called when the thread begins its life (i.e., after calling a constructor of the thread class). The thread is alive and working until the *run()* method completes, or the thread is killed. In FUJABA, the class *AEngine* implements a *run()* method, which calls some other methods of the class *AEngine* that are overridden in its subclasses. The subclasses do not implement their own *run()* method.

When studying the behavior of a single thread, the event trace information needs to be generated for objects (i.e., senders and receivers of method calls were set to be objects). When examining the behavior of the dialog in Figure 2 event trace information for classes was enough; each usage of a dialog represents the life-time of one object, and the dialog was used so that no two objects were alive at the same time. This is not the case with instances of subclasses of *AEngine*. Since subclasses of the *AEngine* class do not even have a *run()* method, not only the objects of one of its subclasses, but all the objects of any of its subclasses might be running their *run()* method simultaneously.

The behavior of an instance of class *AInheritanceCheckerEngine* is studied next. As in Section 5.1, breakpoints were set to the *run()* method of the class *AEngine* and for all methods that have a call dependency with it. Altogether 39 scenario diagrams were generated as a result of the following usage of FUJABA: the application was started, a project was loaded, the class diagram of the project was edited (a couple of methods were edited), the edited class diagram was saved, and FUJABA was exited. From the resulting scenario diagrams it can be noticed that 17 objects of the *AInheritanceCheckerEngine* class were created during the usage. Figure 9 shows a state diagram generated for one of them.

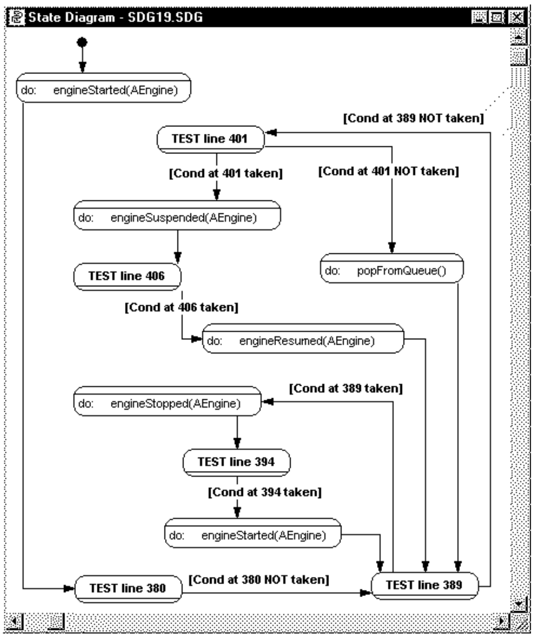

Figure 9. A state diagram for one instance of class AInheritanceCheckerEngine.

Objects and their interactions define the behavior of an object-oriented software system. Because of polymorphism, understanding the behavior is cumbersome. Concluding the behavior of the objects might be impossible even if information about the run-time behavior is generated on the class level (i.e., the objects themselves are not identified). Means to describe the actual behavior of the objects are thus needed. The event trace information generated while running the target system tends to grow rapidly, especially, if the information is generated at the object level. One object may take part in tens of scenario diagrams. Browsing the scenario diagrams can easily get difficult and troublesome. The state diagram synthesis technique provides a quick and efficient way to focus on the overall run-time behavior of a single object.

6. Related research

The need for tools that support dynamic reverse engineering of object-oriented software systems has been acknowledged. We consider here tools, environments, and methods that support understanding the run-time behavior of an object-oriented software system by constructing dynamic views to visualize its execution.

ISVis is a visualization tool that supports the browsing and analysis of execution scenarios [3]. A source code instrumentation technique is used to produce the execution scenarios. To avoid event explosion, the user can select a list of files, which will be instrumented. More fine-grained selections can not be made. In our approach, the user can select the classes and/or a set of methods for which the dynamic event trace information is collected. In ISVis, the event trace can be analyzed using a variation of an MSC called *Scenario View*. Interaction patterns can be found by a variety of pattern matching algorithms in ISVis. The found patterns will be high-lighted on the Scenario View but they can not be used to structure the original event trace. In our approach, behavioral patterns can be searched based on exact string-based matches only. The found patterns are used to structure the set of scenarios generated.

Jinsight is a tool for visualizing the dynamic behavior of Java programs [2]. It views information about object population, method invocations, garbage collection, CPU and memory bottlenecks, thread interactions, and deadlocks. The event traces are produced as a result of instrumenting Java Virtual Machine. Jinsight offers several kinds of views that can be used to analyze the overall object interaction, call stacks of threads, references among objects, and event traces over time. Jinsight uses a tree structured interaction diagram notation that resembles execution pattern views of Ovation [9, 10]. Jinsight is one of the most versatile dynamic reverse engineering tools available. It includes various views for examining different aspects of the run-time behavior. However, state machines, which are commonly used diagrams for specification of the dynamic behavior, are not supported by Jinsight. In addition, Jinsight does not generate information about the dynamic control flow of a selected object or method. These features are included in Shimba.

Walker *et al.* use high-level models for visualizing program execution information [18]. The visualization focuses on object information and interaction information (e.g., a current call stack and a summary of calls). In the main view, called a *cel*, high-level software components are represented as boxes. The interaction among the components is shown by various kinds of directed edges between two boxes. The information is collected by instrumenting the Smalltalk virtual machine to log the events when they occur. The mapping between low-level software artifacts and high-level components they belong to (i.e., boxes in a cel view) is done manually using a declarative mapping language. In Shimba, the run-time information is generated by running the target Java application or applet under a debugger. Algorithms for state diagram synthesis, state diagram optimization, and behavioral pattern recognition are used to build different kinds of abstracted dynamic views. Furthermore, high-level static components can be constructed using Rigi and used to build high-level scenario views that visualize the interaction among the high-level static components [16].

A source code instrumentation technique is used in Scene for producing event traces and visualizing them as scenario diagrams [6]. For compressing the large amount of extracted event trace information Scene shows the operation calls (messages) in a closed form as default: the internal events of a call are not shown unless 'opened' by clicking the call arc. In this way the user can proceed to the interesting level, in a top-down fashion. Corresponding "horizontal abstractions" can be made using execution pattern views of Ovation [10, 9]. Unlike Shimba, Scene does not take advantage of behavioral patterns for structuring and decreasing the information shown in scenario diagrams.

7. Discussion

In this paper, dynamic reverse engineering using Shimba environment has been discussed. Information about static software artifacts and their relations is viewed using the Rigi reverse engineering tool [8]. The dynamic event trace information is generated automatically as a result of running the target system under a customized sdk debugger and viewed as scenario diagrams using the SCED tool [5]. A scenario diagram is a variation of a UML sequence diagram.

SCED scenario diagram notation is an extension of a basic MSC, containing semantically similar concepts as are included in UML sequence diagrams. In most of the current object-oriented modeling approaches (e.g, in UML), the behavioral modeling relies on state machines. This is the case also in SCED. The specification of the behavior of an object is given as a state diagram, which is a variation of a UML statechart diagram. In SCED, state diagrams can be synthesized automatically from scenario diagrams. The dynamic reverse engineering approach discussed in this paper takes advantage of the facility: not only scenario diagrams but also state diagrams are generated automatically based on example usage of a target Java software system. A state diagram provides a different angle for studying the dynamic behavior than scenario diagrams. While scenario diagrams show the sequential interaction among several objects, a state diagram shows the total run-time behavior of a certain object or method, disconnected from the rest of the system. To the best of our knowledge, similar features are not provided by other dynamic reverse engineering tools. Moreover, Shimba is able to generate information about a dynamic control flow of a selected method or an object and to view it as a SCED state diagram.

The dynamic reverse engineering techniques proposed in this paper have a few limitations. First, the string matching algorithms are able to find behavioral patterns only if they lay entirely inside one SCED scenario diagram. Hence, the patterns that begin from the end of one scenario and continue in the next scenario are not recognized. Furthermore, the proposed method is able to find exact matches only. It would be useful to give the user some freedom in defining what is considered to be similar behavior. Second, modeling the overall behavior of a complicated object might result in a large statechart diagram that does not fit entirely in the SCED window in a readable form. The statechart diagram optimization algorithms are able to decrease the size of the statechart diagram dramatically. However, some of the synthesized statechart diagrams generated in the case study were too large even after the optimization. To increase the readability of the statechart diagram window, SCED allows the user to scroll and zoom it. Third, naming of state and assertion boxes, which are used to capture dynamic control flow information, is not very informative nor useful. Since Shimba is independent of the source code of the target software system, the state boxes are not labeled by the actual conditional statements, even though that would be more descriptive.

Understanding the overall behavior of objects that play a significant role in an object-oriented software system is often essential. Such an object could be a user interface object or a middleware object in an internet-based distributed object system. A distributed object could be, for instance, an object that supports object interoperability using technologies like CORBA, RMI, DCOM, etc. The interaction among objects in a distributed object system is managed through specific interfaces provided by the used technology. This interaction could be visualized using scenario diagrams. The state diagram synthesis technique provides a powerful way to examine the overall behavior and the dynamic control flow of such objects.

Acknowledgements

I would like to thank Jens Jahnke, Kai Koskimies, Hausi Müller, and Erkki Mäkinen for commenting and proofreading the paper. This research has been financially supported by the Academy of Finland, the Center for Technological Development in Finland (TEKES), Nokia Research Center, Nokia Networks, Neles Automation, Sensor Software Consulting, Acura Systems, and Plenware.

References

[1] R. Boyer and J. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[2] IBM Research. Jinsight, visualizing the execution of java programs. [http://www.research.ibm.com/ jinsight/], 2000.

[3] D. Jerding and S. Rugaber. Using visualization for architectural localization and extraction. In *Proc. of the 4th WCRE*, pages 56–65, 1997.

[4] K. Koskimies and E. Mäkinen. Automatic synthesis of state machines from trace diagrams. *Software—Practice & Experience*, 24(7):643–658, 1994.

[5] K. Koskimies, T. Männistö, T. Systä, and J. Tuomi. Automated support for modeling oo software. *IEEE Software*, 15(1):87–94, January/February 1998.

[6] K. Koskimies and H. Mössenböck. Scene: Using scenario diagrams and active text for illustrating object-oriented programs. In *Proc. of the 18th ICSE*, pages 366–375, 1996.

[7] D. B. Lange and Y. Nakamura. Object-oriented program tracing and visualization. *IEEE Computer*, 30(5), 1997.

[8] H. Müller, K. Wong, and S. Tilley. Understanding software systems using reverse engineering technology. In *The 62nd ACFAS*, 1994.

[9] W. D. Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. In *Proc. of the 10th OOPSLA*, pages 326–337, Oregon, USA, 1994.

[10] W. D. Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *Proc. of the COOTS*, 1998.

[11] Rational Software Corporation. The unified modeling language notation guide v.1.3. [http://www.rational.com], 2000.

[12] I. Rockel and F. Heimes. Fujaba - homepage. [http://www.uni-paderborn.de/fachbereich/AG/schaefer/ ag_dt/PG/Fujaba/fujaba.html], February 2000.

[13] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Reference Manual*. Addison-Wesley, 1999.

[14] Sun Microsystems Inc. Java(tm) 2 sdk, standard edition 1.2. [http://java.sun.com/products/jdk/1.2/], 2000.

[15] T. Systä. Automated support for constructing omt scenarios and state diagrams in sced. Unversity of Tampere, Dept. of Computer Science, Report A-1997-8, 1997. Licentiate's thesis.

[16] T. Systä. On the relationship between static and dynamic models in reverse engineering java software. In *Proc. of the 6th WCRE*, pages 304–313, 1999.

[17] T. Systä. Static and dynamic reverse engineering techniques for java software systems. Unversity of Tampere, Dept. of Computer Science, Report A-2000-4, 2000. PhD dissertation.

[18] R. Walker, G. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak. Visualizing dynamic software system information through high-level models. In *Proc. of the 14th OOPSLA*, pages 271–283, 1998.