SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2001; **31**:371–394

Shimba—an environment for reverse engineering Java software systems‡

Tarja Systä[1,*,†], Kai Koskimies[1] and Hausi Müller[2]

[1]*Software Systems Laboratory, Tampere University of Technology, P.O. Box 553, FIN-33101 Tampere, Finland*
[2]*Department of Computer Science, University of Victoria, P.O. Box 3055, Victoria, BC, Canada V8W 3P6*

SUMMARY

Shimba is a reverse engineering environment to support the understanding of Java software systems. Shimba integrates the Rigi and SCED tools to analyze and visualize the static and dynamic aspects of a subject system. The static software artifacts and their dependencies are extracted from Java byte code and viewed as directed graphs using the Rigi reverse engineering environment. The run-time information is generated by running the target software under a customized SDK debugger. The generated information is viewed as sequence diagrams using the SCED tool. In SCED, statechart diagrams can be synthesized automatically from sequence diagrams, allowing the user to investigate the overall run-time behavior of objects in the target system.

Shimba provides facilities to manage the different diagrams and to trace artifacts and relations across views. In Shimba, SCED sequence diagrams are used to slice the static dependency graphs produced by Rigi. In turn, Rigi graphs are used to guide the generation of SCED sequence diagrams and to raise their level of abstraction. We show how the information exchange among the views enables goal-driven reverse engineering tasks and aids the overall understanding of the target software system. The FUJABA software system serves as a case study to illustrate and validate the Shimba reverse engineering environment. Copyright © 2001 John Wiley & Sons, Ltd.

KEY WORDS: software reverse engineering; reverse engineering environment; program comprehension; SCED; Rigi; Java

*Correspondence to: Tarja Systä, Software Systems Laboratory, Tampere University of Technology, P.O. Box 553, FIN-33101 Tampere, Finland.
†E-mail: tsysta@cs.tut.fi

‡This paper is based on (1) 'On the relationships between static and dynamic models in reverse engineering Java software' by T. Systä which appeared in the *Proceedings of WCRE 1999*, Atlanta, Georgia, USA, IEEE Computer Society Press, 1999, pp. 304-313, and on (2) 'Understanding the Behavior of Java programs' by T. Systä which appeared in the *Proceedings of WCRE 2000*, Brisbane, Australia, IEEE Computer Society Press, 2000, pp. 214–223.

Contract/grant sponsor: Academy of Finland
Contract/grant sponsor: National Technology Agency of Finland (TEKES)
Contract/grant sponsor: Nokia
Contract/grant sponsor: Neles Automation
Contract/grant sponsor: Sensor Software Consulting
Contract/grant sponsor: Acura Systems
Contract/grant sponsor: Plenware

Received 15 September 2000
Revised 30 November 2000
Accepted 8 January 2001

INTRODUCTION

The need for maintaining, reusing, and re-engineering existing software systems has increased dramatically over the past decade. Changed requirements or the need for software migration, for example, necessitate renovations for business-critical software systems. Reusing and modifying long-lived legacy systems involves complex and expensive tasks because of the time-consuming process of program comprehension. Many reverse engineering tools have been built over the last 15 years to help the comprehension of software systems. These tools aid the extraction of software artifacts and their dependencies and the synthesis of high-level concepts. Moreover, these tools provide support for analyzing software systems and automate mundane and repetitive program understanding operations.

Many of the currently available reverse engineering and design recovery environments emphasize *static reverse engineering* (i.e. analyzing the static structure of a subject software system). Such environments have been successfully used to support the understanding of applications written in procedural programming languages. However, the change of programming languages and styles also affects the evolution of reverse engineering tools and methods. The adoption of the object-oriented programming paradigm has changed programming styles significantly. The dynamic nature of object-oriented programs emphasizes the importance of *dynamic reverse engineering* (i.e. analyzing the run-time behavior of a subject software system). To understand the architecture of an object-oriented software system, both static and dynamic analyses are needed.

One of the most challenging tasks in reverse engineering is to build descriptive and readable views of the software at the right level of abstraction. The extracted information can be merged into a single view and support for information filtering techniques and means to build abstractions can be provided to keep the view manageable and understandable. However, when both static information and dynamic information are considered, the chosen view often serves either the static or the dynamic aspects effectively—but rarely both. In practice, the dynamic information is just viewed against a previously built static model. Another approach is to construct separate dynamic and static models. Since this is common practice in forward engineering, it is natural to follow this approach also in reverse engineering. As in forward engineering, having separate views requires meaningful, consistent, and effective integration.

Shimba, a prototype reverse engineering environment, has been built to support the understanding of Java software systems. Shimba supports the exploration, visualization, and analysis of both static and dynamic views of the system. Shimba aims at combining static and dynamic reverse engineering techniques and carrying them out in parallel to exploit the extracted information more effectively [1]. Both static and dynamic information concerns software artifacts and their relations. The shared information enables information exchange among the models. This paper shows how goal-driven reverse engineering and general program understanding tasks are supported by Shimba.

In particular, we discuss the static and dynamic reverse engineering capabilities of the Shimba environment. To validate the usefulness of the approaches presented in this paper, we performed a case study on FUJABA, a 700 class Java software system [2]. We investigated and evaluated both general program understanding and goal-driven reverse engineering tasks.

OVERALL DESCRIPTION OF SHIMBA

In Shimba, the static information is extracted from Java byte code using a tool called *JExtractor*. The extracted information is visualized and analyzed with the Rigi reverse engineering tool [3]. Rigi uses a

graph model to represent information about software entities, relationships, attributes, and abstractions over them [4]. The dynamic event trace information is generated automatically as a result of running the target system under *JDebugger*, a customized *SDK* debugger [5]. The event trace can be viewed as sequence diagrams using the dynamic analysis engine *SCED* [6]. Optionally, information about the dynamic control flow of selected objects can be extracted and added to the sequence diagrams generated. A SCED sequence diagram corresponds to a sequence diagram in the *Unified Modeling Language (UML)* [7,8]. Many other dynamic reverse engineering tools use variations of *Message Sequence Charts (MSCs)* [9] to visualize the run-time behavior of the target object-oriented software system [10–14].

In Shimba, the visualization of the run-time behavior of object-oriented software has been taken one step further—not only the sequence diagrams, but also the complete model of the dynamic behavior (i.e. a statechart diagram) can be built automatically as a result of the execution of a target system [15]. The automated statechart diagram synthesis feature of SCED allows this innovative step [6,16]. The SCED statechart diagram notation can be characterized as a simplified UML statechart diagram notation. Generated statechart diagrams allow the user to examine the dynamic behavior from a different perspective compared to sequence diagrams. While sequence diagrams show the interaction among several objects in example runs, a statechart diagram shows the overall behavior of a certain object in the target system. Understanding the overall behavior of certain key objects is often essential for understanding purposes. Such objects could be user interface objects or interoperable middleware objects in an internet-based distributed object system.

Both Rigi views and SCED sequence diagrams contain information about software artifacts and their relations. In Shimba, this information is used as a basis for information exchange between the two systems [1]. For example, the user can select certain software artifacts from a static Rigi view and generate run-time information concerning the usage of only those artifacts with SCED. For debugging purposes, the user can take a set of sequence diagrams that characterize the exceptional behavior and *slice* the corresponding Rigi graph with them. As a result, the user obtains a static view that can be used to analyze how the part that caused the undesired behavior was constructed. We call this *model slicing* (i.e. one model is sliced by means of information included in a set of other models).

In addition to goal-driven reverse engineering tasks, Shimba supports general understanding of Java software systems. In Rigi, graphs can be nested to view the software at different levels of abstraction. The user can identify clusters of nodes (related according to selected criteria) and collapse them into nodes that represent higher level components (e.g., subsystems or packages). The new graph, thus, represents a more abstract view of the software, containing fewer nodes and arcs. The nodes can be clustered to an arbitrary depth. In Shimba, the level of abstraction of SCED sequence diagrams can be raised using a high-level Rigi graph. The result is a sequence diagram that shows the interaction among the high-level static components. The level of abstraction of the sequence diagrams can also be raised by structuring them with behavioral patterns (i.e. repeated similar behavior). Shimba provides string matching algorithms to recognize behavioral patterns in the sequence diagrams. Furthermore, a Rigi graph can be sliced with SCED sequence diagrams. This feature can be used for studying the structure of the parts of the target software that caused the behavior depicted in the sequence diagrams. Dynamic information can also be used to annotate the Rigi graph with weight values that indicate how many times different parts of the software have been used during the execution. Figure 1 shows the major components and their interactions of Shimba.

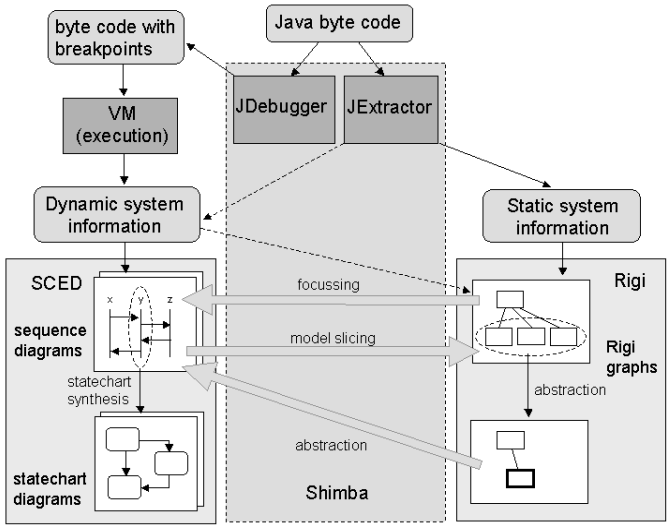

Figure 1. Shimba—An environment for reverse engineering Java software.

In most reverse engineering environments, the static information is parsed from the source code. Shimba, in contrast, extracts artifacts from the *Java byte code* representation of a program. We had three main reasons for choosing Java byte code. First, extracting the static information from byte code allows Shimba to be independent of the source code, which is often not even available. Second, since the same byte code is used for both static and dynamic analysis, the user can be confident that the constructed models do not contain any inconsistencies due to different versions of the target software. Third, extracting information from Java byte code is straightforward and efficient. The downside of using byte code only is the loss of some valuable pieces of information. For instance, comments and conditions of branching statements are lost.

STATIC REVERSE ENGINEERING USING RIGI

Rigi is an interactive and visual reverse engineering system [17]. A semi-automatic reverse engineering approach of Rigi consists of two phases: the identification of software artifacts and their relations and the extraction of design information and system abstractions [18]. Rigi includes several parsers to extract artifacts from the source code of a target software. The extracted static information can be visualized and analyzed using the Rigi visualization engine *Rigiedit* [18].

In Shimba, Rigi has been customized to analyze Java software systems. The *JExtractor* tool extracts selected components and dependencies to form a simple static model of a subject system. In particular, it extracts the following artifacts from Java byte code: *classes, interfaces, methods, constructors,*

variables, and *static initialization blocks*. In addition, some attribute values are attached to these software artifacts. They contain information about *return types*, *visibility*, and other *access modifiers*. Furthermore, the JExtractor tool extracts the following relationships among these artifacts: *extension* (i.e. a class extends another class), *implementation* (i.e. a class implements an interface), *containment* (i.e. a class contains a method), *call* (i.e. a method calls another method), *access* (i.e. a method accesses a variable), and *assignment* (i.e. a method assigns a value to a variable).

The extracted information is visualized using Rigi dependency graphs. Rigi uses directed graphs to view such static information—the software artifacts and their relations are represented as nodes and arcs, respectively. Colors signify node and arc types. Attribute values, attached to nodes or arcs, are usually hidden, but can be browsed using the attribute dialog window.

Shimba provides facilities to compute the Chidamber and Kemerer suite of object-oriented metrics [19]. The metrics measure properties of the classes, the inheritance hierarchy, and the interaction among classes of a subject system. Since Shimba is primarily intended for the analysis and exploration of Java software, the metrics have been tailored to measure properties of software components written in Java [20]. The following measures can be added as attribute values to software artifacts: *Cyclomatic Complexity (CC)*, *Weighted Methods per Class (WMC)*, *Lack of Cohesion (LCOM)*, *Response For a Class (RFC)*, *Coupling Between Objects (CBO)*, *Depth of Inheritance Tree (DIT)*, and *Number of Children (NOC)*. These metrics are useful when understanding software systems using a reverse engineering environment such as Shimba. The static dependency graphs of the system under investigation are decorated with measures obtained by applying the object-oriented metrics to selected software components. Shimba provides tools to examine these measures, to find software artifacts that have values that are in a given range, and to detect correlations among different measures. The object-oriented analysis of the subject Java system can be investigated further by exporting the measures to a spreadsheet.

Dynamic modeling using SCED

The SCED tool has been built to support the dynamic modeling of object-oriented applications [6]. Most user interaction with SCED involves two independent editors: a *sequence diagram editor* and a *statechart diagram editor*. In SCED, the basic MSC notation [9] has been extended with new concepts, including *action boxes*, *assertion boxes*, *state boxes*, *conditional constructs*, *repetition constructs*, and *subscenarios*. Action boxes are used to visualize a sent message that is received by the object itself. The usage of assertion boxes in Shimba is associated to the usage of state boxes. They are generated to indicate which clauses of conditional statements are executed. The behavioral patterns in the event trace are visualized with repetition constructs and subscenarios. A subscenario is shorthand notation for inserting the contents of the referred sequence diagram at the position of the subscenario box, thus corresponding to a MSC reference [9]. Conditional and repetition constructs as well as subscenarios can be nested to an arbitrary depth. The SCED sequence diagram notation is shown in Figure 2.

For most of the SCED sequence diagram elements there are corresponding elements in the UML sequence diagram notation [7,8]. For example, SCED *assertions*, *conditional constructs*, and *repetition constructs* correspond to UML *constraints*, *conditional branching*, and *iteration*, respectively. However, UML sequence diagram notation does not include a notational device that corresponds to a SCED subdiagram box, which provides nesting of sequence diagrams. Moreover, UML sequence diagrams have no convenient tool for modeling a repetition of subsequent, disconnected

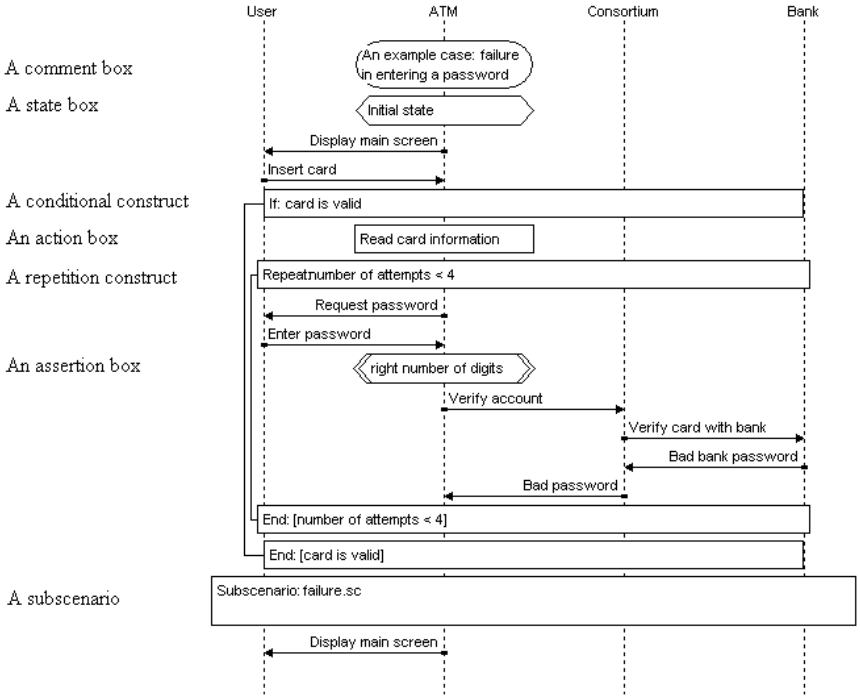

Figure 2. SCED sequence diagram notation concepts.

messages. Because of repetition constructs and subscenarios, SCED sequence diagrams are more effective for modeling behavioral patterns than UML sequence diagrams.

The SCED statechart diagram notation can be characterized as simplified UML statechart notation. The main restriction is that SCED statechart diagrams can not be used to express concurrency, i.e. *composite states* with concurrent substates. Also, *history states*, *submachine reference states*, and *timing constraints* are not supported by SCED statechart diagram notation.

Koskimies and Mäkinen have demonstrated how a minimal state machine can be synthesized automatically from trace diagrams [21]. We implemented this algorithm, with few modifications, for SCED to provide statechart diagram synthesis for a selected object from a set of SCED sequence diagrams [16]. The algorithm first extracts a trace from the sequence diagrams by traversing the vertical line of an object from top to bottom in each sequence diagram. The algorithm then maps elements in the event trace to transitions and states in a statechart diagram. A sent message is considered as a primitive action performed by an object. A received message, in turn, causes an object to react (and possibly send another message). Thus, the algorithm maps each sent message with an action associated with a state and each received message with a transition. The additional SCED sequence diagram concepts do not cause major changes to the basic algorithm. Assertion boxes are treated like received messages,

and action boxes like sent messages. Algorithmic constructs (i.e. conditional and repetition constructs) and subscenarios are handled recursively.

The user can select one participant in a sequence diagram and synthesize a statechart diagram automatically for it. This synthesis can be done for one sequence diagram only or for a specified set of sequence diagrams. The statechart diagram synthesis algorithm works incrementally—sequence diagrams can be synthesized into an existing statechart diagram. Moreover, statechart diagrams can be synthesized for single methods.

The synthesis algorithm of SCED produces a rudimentary statechart diagram with at most one action in a single state. Thus, SCED provides *optimization* algorithms for transforming a synthesized and/or edited statechart diagram into a more compact form while preserving its information [15]. The optimization algorithms detect similar responses to certain messages and use that information to restructure the statechart diagram. The statechart diagram is modified by adding UML statechart notation elements into it. The generated elements include *entry actions*, *exit actions*, actions fired by *internal transitions*, and actions attached to transitions.

Collecting dynamic information

To collect dynamic information using Shimba, one selects the classes and/or methods for which the event trace information is to be collected. One can also choose exceptions to be tracked. Event trace information is generated (i.e. new elements are added to SCED sequence diagrams) when methods are called or exceptions are thrown. To capture the event trace information, breakpoints are set with the JDebugger tool (e.g., at the first line of a set of selected member functions or seleccted/all member functions of selected/all classes). The dynamic control flow information is also generated using breakpoints at control statements (i.e. *if, for, while, do-while*, and *switch-case* statements). Event trace information is generated when a breakpoint is hit or an exception is thrown. The JDebugger tool keeps a stack of previously activated class members. The stack information can be used to infer the sender of the message generated when a breakpoint is hit.

The control statements indicate branching points in the control flow. To capture the dynamic control flow information, state boxes and assertion boxes are added to the SCED sequence diagrams corresponding to the conditions to be evaluated and the resulting values, respectively. For example, a state box '*TEST line 311*' in a SCED sequence diagram in Figure 5 indicates that a binary instruction, which corresponds to a Boolean expression at line 311 in the source code, is executed. An assertion box '*Cond at 311 taken*', in turn, indicates that the binary instruction yielded true during execution. Note that Shimba does not rely on the availability of the source code of the target software system. Hence, state boxes are not labeled by the actual Boolean expressions, even though that would be more descriptive. By giving source line references (available in the byte code if the source code has been compiled using a debugging switch), we at least want to provide the user with the opportunity to quickly find the relevant lines in the source code.

APPLICABILITY OF SHIMBA

Because of the strong dependence between the structural and behavioral aspects of a software system, the static and dynamic analyses of a subject system should be coupled or integrated. Most currently

available reverse engineering tools provide few facilities to support coupling between static and dynamic analyses. The tools either concentrate on static or dynamic reverse engineering, but usually not both. The tools that support both either merge the extracted information into a single view or construct and analyze static and dynamic views separately. Neither of these approaches is particularly satisfactory. Thus, we decided to follow an innovative approach in the Shimba environment combining by static and dynamic analyses at early stages of the reverse engineering process.

Shimba supports both overall understanding of the target software system and goal-driven reverse engineering tasks. To achieve the former task, Shimba provides automated tools that can be used to find answers to various questions including:

1. What are the static software artifacts and how are they related?
2. How are the software artifacts used at run-time?
3. What is the high-level structure of a subject system?
4. How do the high-level components interact with each other?
5. Does the run-time behavior contain regular behavioral patterns that are repeated? If so, what are the patterns and under which circumstances do they occur?
6. How heavily has each component of a subject system been used at run-time and which components have not been used at all?

Goal-driven reverse engineering approaches are useful for debugging purposes, performance analyses, requirements tracing, or evaluating a user interface component. Shimba supports goal-driven reverse engineering by offering automated tools to analyze exceptional behavior and to answer questions about a target software system of the following sort:

1. How does a certain component behave and how is it related to the rest of the system?
2. When was an exception thrown or when did an error occur? What happened before that and in which order?
3. How is the component that causes exceptional behavior constructed?

Even a relatively brief usage of the target software system typically produces a large amount of event trace information. In Shimba, the event trace information is stored as a set of SCED sequence diagrams. Like other dynamic reverse engineering tools [11,14,22], Shimba provides means to search for behavioral patterns over the event trace information. However, it is difficult to understand the overall behavior of a single object by examining the sequence diagrams or the behavioral patterns. The SCED statechart diagram synthesis technique greatly helps in alleviating this problem. Shimba allows the user to study the total behavior of an object or a method (based on the run-time usage) as one model, disconnected from the rest of the system. The statechart diagrams can be used to find answers to questions of the following kind:

1. What is the dynamic control flow and the overall behavior of an object or a method?
2. How can a certain state of an object be reached (i.e. which execution paths lead from the initial state to this state) and how does the execution continue (i.e. which execution paths lead from this state to the final state)?
3. To which messages has an object responded at a certain state during its lifetime?
4. Which methods of the object have been called during execution?

In what follows, we discuss selected reverse engineering approaches supported by Shimba and results of a case study with the FUJABA system (version 0.6.3-0), which is freely available and downloadable from the Web [2]. The primary objective of the FUJABA project and environment is *Round Trip Engineering* using the UML, Story Driven Modeling (SDM), Design Patterns, and Java. The implementation of FUJABA 0.6.3-0 consists of about 700 Java classes and provides a variety of tools including editors for defining both structural (i.e. class diagrams) and behavioral (i.e. activity diagrams, UML activity/story diagrams) aspects of a software system. In the case study, we focused on studying the functionality of the FUJABA class diagram editor.

USING STATIC INFORMATION TO GUIDE THE GENERATION OF DYNAMIC INFORMATION

One of the main strengths of Shimba is the capability to guide and manage the generation of dynamic information using information gleaned from static analyses. When reverse engineering an existing, unknown software system, the engineer might begin the investigation with a specific component. If the engineer is only interested in the dynamic behavior of a specific component of the subject system, it is not meaningful to generate information for all object interactions of the system, but only those interactions that have an effect on the behavior of that particular component. Such information prefiltering can drastically reduce the size of the generated event traces while focusing on the information of interest.

Rigi provides many Tcl/Tk scripts for analyzing static graphs. Moreover, the engineer can easily extend the Rigi script library to provide domain specific or custom functionality. In Shimba, the information filtering mechanisms of Rigi can be used to guide the JDebugger tool to restrict the amount of event trace information generated. After defining the components of interest in a Rigi graph, the user can select all the *Method*, *Constructor*, and *Staticblock* nodes (representing member functions) for which the JDebugger tool will set breakpoints.

Assume that the user is interested in the behavior of a set S_o of classes. Let S be the transitive closure of S_o with repsect to the inheritance relation. Furthermore, let M be the set of all members of all classes in S, that is, for each $m \in M$, there is a *contains* arc from one $s \in S$ to m. Finally, let M_t be a set of nodes to which there is a call arc from any node in $S \cup M$, and let M_s be a set of nodes from which there is a call arc to any node in $S \cup M$. The sets M_t and M_s contain member functions that might be interacting with member functions in M. It is useful then to filter out everything from the Rigi graph except the following nodes: all classes and interfaces in S, all members in $M_s \cup M_t$, and all nodes that represent overridden methods of methods in M_t.

USING THE STATECHART DIAGRAM SYNTHESIS FACILITY OF SCED FOR DYNAMIC REVERSE ENGINEERING

Figure 3 shows a dialog box used in FUJABA for defining and editing parameters of methods. To understand the behavior of FUJABA, let us begin by examining the internal behavior and the run-time control flow of the *modify* button in the dialog window. Each time the *modify* button is pressed, method *modifyButton_actionPerformed(ActionEvent)* of the dialog class *PEParameters* is called. By running

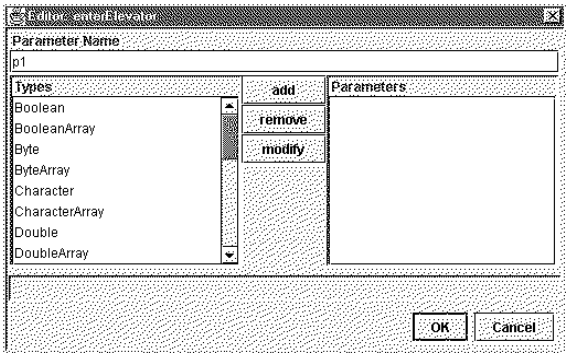

Figure 3. A dialog used by the FUJABA class diagram editor.

a few Rigi scripts, 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 4. From the dependencies 11 method nodes and three 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 immaterial for understanding the behavior of the dialog in Figure 3. Furthermore, it can be easily certified (by running a couple of Rigi scripts) that none of the methods that are called by the method *modifyButton_actionPerformed (ActionEvent)* are overridden in any subclass. Nodes that represent the overridden methods should naturally be included in the dependency graph.

Finding the right nodes using Rigi is simple, since the package and class declaration are included in the names of all nodes. Hence, the nodes can be found (e.g., for clustering or filtering purposes) using a simple grepping script. On the other hand, because of the used naming convention, the names of nodes in Rigi graphs tend to be long and overlapping. This can be seen, for example, from Figures 3 and 10. In Rigi, the user is provided a possibility to show or hide the names of selected/all nodes whenever she wants.

The debugger sets breakpoints automatically for the methods and constructors visualized in Figure 4 (selected nodes). According to the user's instructions, breakpoints were set to capture both object interaction and dynamic control flow information. The dialog in Figure 3 was used four times in the following way: (1) the name of the selected parameter was changed and the *modify* button was pressed; (2) the *modify* button was pressed when none of the parameters were selected; (3) the type of the selected parameter was changed and the *modify* button was pressed; and (4) the name of the selected parameter was deleted and the *modify* button was pressed. Figure 5 shows the sequence diagram resulting from the first case. Figure 6 depicts the merged internal behavior of the method

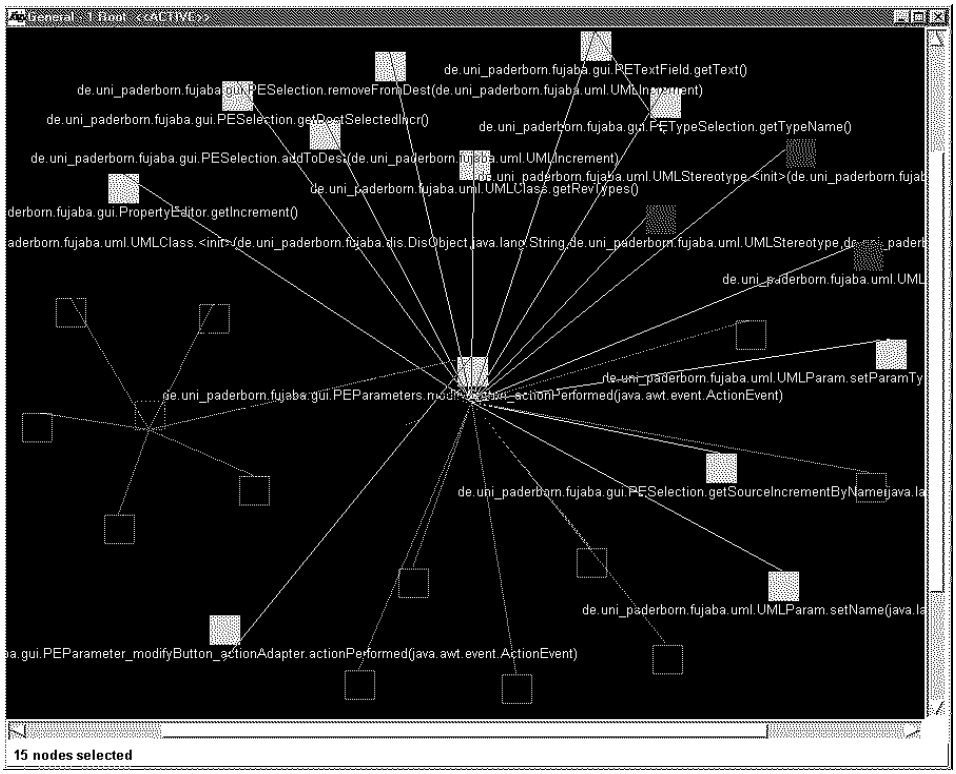

Figure 4. A Rigi graph that shows the method *modifyButton_actionPerformed(ActionEvent)* and its dependencies.

modifyButton_ actionPerformed(ActionEvent) of class *PEParameters*, synthesized from the generated four sequence diagrams.

When generating dynamic information for the whole dialog in Figure 3, the dialog was used three different times while running FUJABA (i.e. three instances of the class *PEParameters* were created), but the objects were not alive at the same time. The usage of the dialog resulted in 20 sequence diagrams. The statechart diagram describing the dynamic control flow of the dialog contains 48 states. Such a large statechart diagram is difficult to read and visualize. Although a relatively simple and brief usage of the system produces a large statechart diagram, it can be assumed that the more the dialog is used, the less the synthesized statechart diagram will grow—rather than more states, more transitions would be generated describing different paths through the statechart diagram.

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

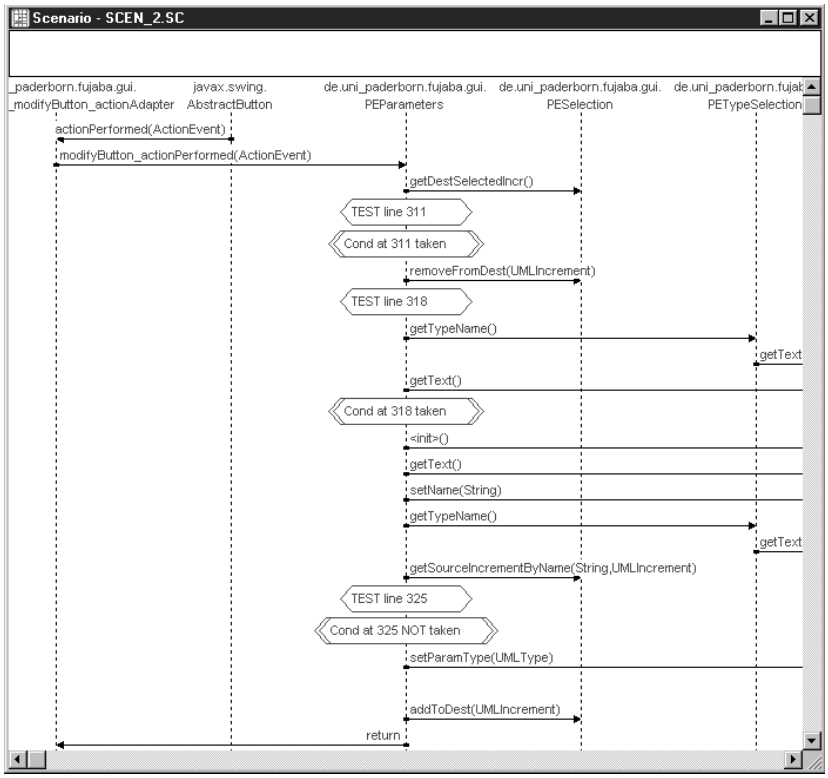

Figure 5. A sequence diagram describing one execution path through the method *modifyButton_actionPerformed(ActionEvent)* of class *PEParameters*.

run() method is called when the thread begins its life. 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 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. the senders and receivers of method calls were set to be objects). When examining the behavior of the dialog in Figure 3, event trace information for classes was sufficient; 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.

For studying the behavior of an instance of class *AInheritanceCheckerEngine*, breakpoints were set to the *run()* method of the class *AEngine* and for all methods that have a call dependency with it.

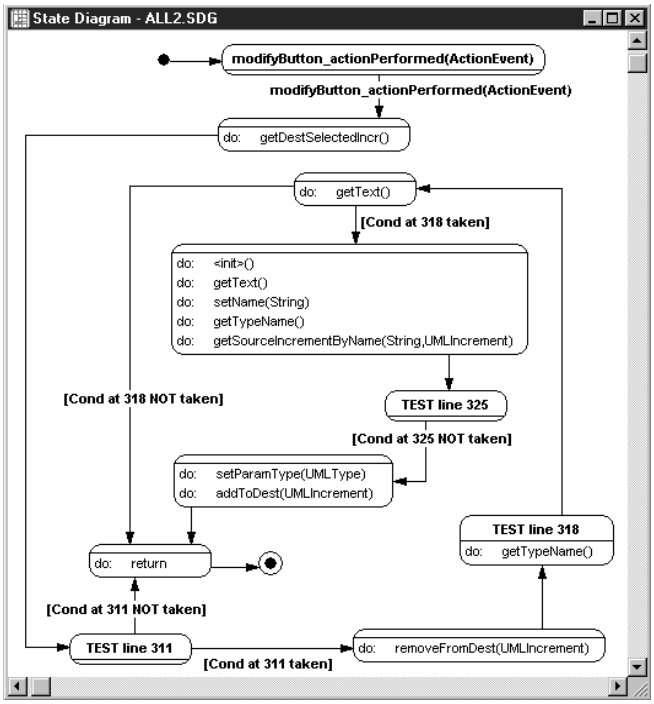

Figure 6. The detailed behavior of the method *modifyButton_action-Performed(ActionEvent)* of class *PEParameters*.

Altogether 39 sequence diagrams were generated as a result of the following usage of FUJABA: (1) the application was started, a project was loaded; (2) 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 sequence diagrams, we infer that 17 objects of the *AInheritanceCheckerEngine* class were created during the usage. Figure 7 shows a state diagram generated for one of them.

Objects and their interactions define the behavior of an object-oriented software system. Because of polymorphism, understanding the behavior is cumbersome. As was seen in the case of the threads in FUJABA, concluding the behavior might require that the individual objects are 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 sequence diagrams. Browsing the sequence diagrams can easily become difficult and troublesome. The statechart diagram synthesis approach provides a quick and efficient way to focus on the behavior of a single object.

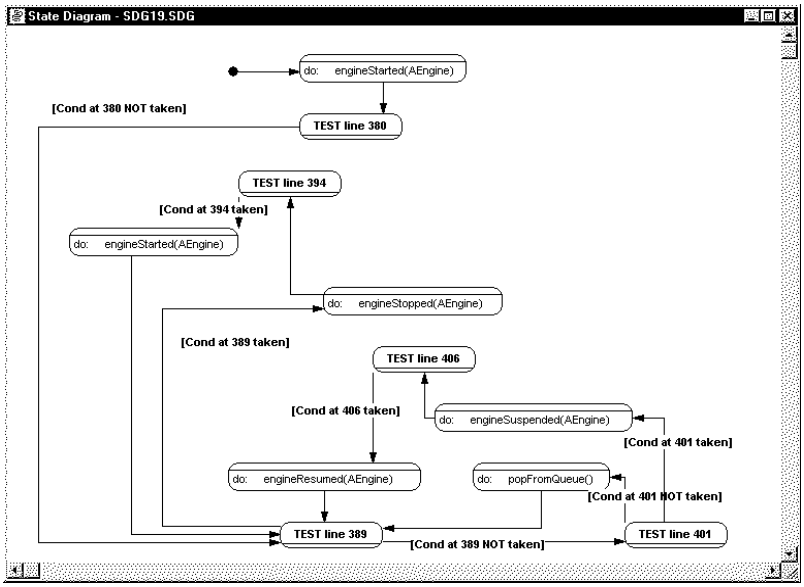

Figure 7. A statechart diagram for an instance of class *AInheritanceCheckerEngine*.

MANAGING THE EXPLOSION OF THE EVENT TRACE

Shimba provides facilities to modify sequence diagrams using the Boyer–Moore string matching algorithms [23]. The identified patterns are shown either as repetition constructs or as subscenarios in SCED sequence diagrams. The 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 a sequence 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 moving a mouse, clicking a mouse, or giving keyboard commands. In such situations a single message might be repeated a huge number of times. Hence repetition constructs for repeated single messages are generated already during the debugging process.

Behavioral patterns may also occur separated from each other. For example, in a graphical user interface, opening of a dialog box causes an initialization sequence that is repeated every time a dialog is opened. These patterns may appear in different sequence diagrams, or at least disconnected from each other. Subscenarios are used to identify such patterns. Subscenario boxes provide a powerful way to link sequence diagrams: instead of repeating the message sequence in a place of their appearance, a subscenario box is used to refer to another sequence diagram that actually contains message sequences.

In FUJABA, the dialog in Figure 3 is called by another dialog that is used to define and edit methods. The class that implements this dialog is called *PEMethod*. To analyze the behavior of this dialog

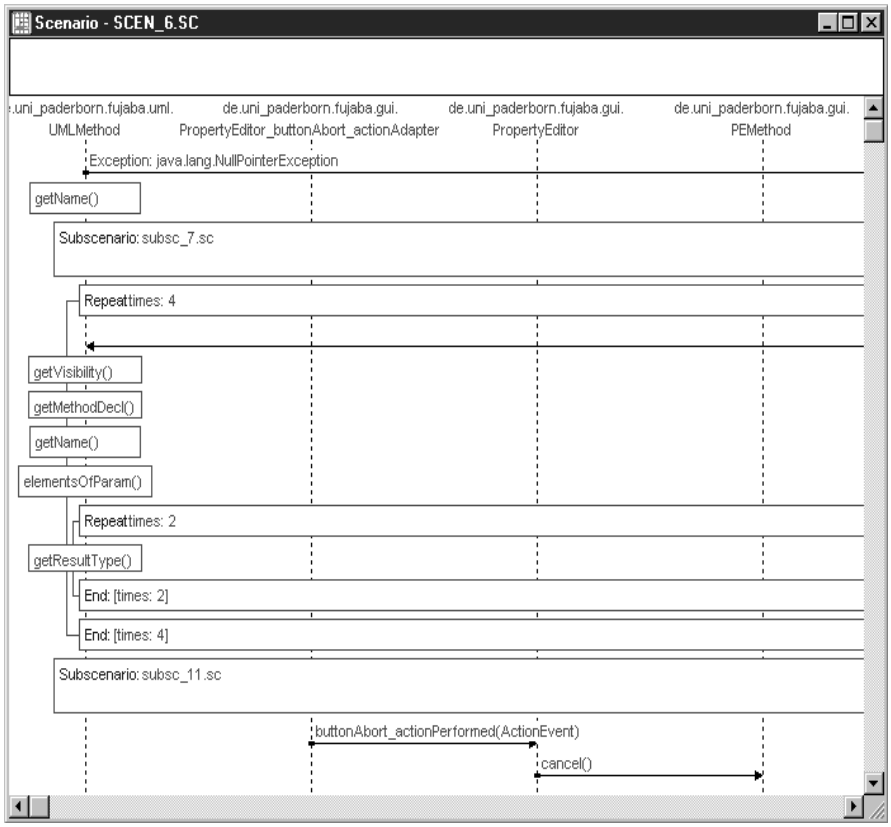

Figure 8. A sequence diagram after applying the scenario structuring algorithm.

and thus study 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 sequence diagrams. The algorithms were applied for six sequence 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 8 shows one of the sequence diagrams after modification 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 sequence diagram element was included in the original sequence diagram. The

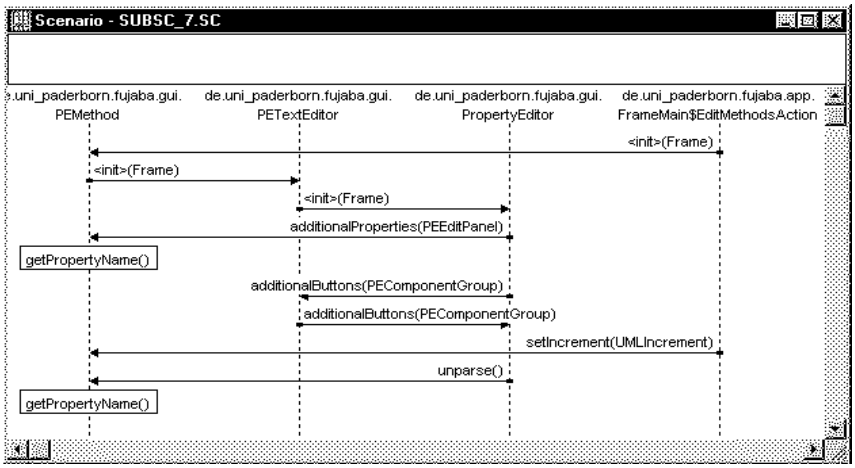

Figure 9. The contents of the subdiagram box *subsc_7.sc* in Figure 8.

sequence diagram depicts the object interaction that was involved when the dialog was opened for a class that has four methods, and exited by pressing the *cancel* button.

The object interaction that is needed for the initialization of the dialog is repeated every time the dialog is opened. The scenario structuring algorithm recognized that pattern and formed the subscenario box *subsc_7.sc* for it. Figure 9 shows the contents of the subscenario box *subsc_7.sc* (<*init*> messages represent constructor invocations). The original sequence diagrams consist of 473 elements. After applying the scenario structuring algorithm the total number of sequence diagram elements was reduced to 201 (i.e. a 58% reduction). The proportion of decreased elements naturally varies depending on the set of sequence diagrams.

Another way to manage huge event traces is to split them into several shorter, more manageable ones. This approach is used also in Shimba. The event trace is split automatically into several sequence diagrams in order to limit the size of a single sequence diagram. At any point during the execution, the user can dump the current event trace into a sequence diagram file and initialize the event trace with a single mouse click. In this way, the user can 'record' the desired execution interval.

SLICING RIGI VIEWS USING SCED SEQUENCE DIAGRAMS

In addition to methods called at run-time, SCED sequence diagrams identify the owner objects (or their classes) as well as the calling objects (or their classes). However, the method is always called from another member function. The static Rigi graph shows a *call* dependence between two methods rather than between the classes of a calling object and a called method. Thus, when slicing the purely static Rigi graph based on example sequence diagrams, nodes that represent methods and classes and

have been visited during the execution, as well as arcs connecting them, are included in the slice. The rest of the graph is filtered out. These filtering algorithms can be run from Rigi. The proposed model slicing approach can be used for studying the parts of the target software that are involved in a specific kind of usage.

The class *PEMethod* (which implements the dialog used for editing methods) had a known bug in its functionality (which has been fixed in the later versions of FUJABA)—if the user selected a method, for which parameters had been given, and pressed the *modify* button, the parameters disappeared. This bug is listed in the FUJABA bug report [2]. To find the source of this bug, breakpoints were set for all methods of the following classes: *UMLClass*, *UMLMethod*, *UMLParam*, and the dialog class *PEMethod* itself. The following actions were executed to generate the run-time information: (1) opening the dialog, (2) entering a new parameter for a selected method using the dialog in Figure 3 (implemented as class *PEParameters*), and (3) pressing the *modify* button. By studying the resulting sequence diagrams, we found that method *addToParam(UMLParam)* of class *UMLMethod* was called by class *PEParameters* when parameters were added for a method. The constructor invocation *<init> ()* of class *de.uni_paderborn.fujaba.uml.UMLMethod* in one of the later sequence diagrams indicates that when the *modify* button was pressed (i.e. *modifyButton_actionPerformed(ActionEvent)* method was called), a new method was created, which suggests that the old method was replaced by a new one.

To examine the structure of the components of FUJABA that might contain the source of the bug, the initial Rigi graph representing the whole FUJABA system was sliced based on a set of sequence diagrams that were generated after the dialog for editing methods was opened. There were altogether 29 such sequence diagrams. The resulting Rigi graph is shown in Figure 10. The node representing method *modifyButton_actionPerformed(ActionEvent)* of class *PEMethod* can be seen at the bottom of the graph. The nodes, whose names are shown, represent methods that could be called from it directly or indirectly. Such a chain of method calls can be easily found by running a simple Rigi script on the graph. The default constructor of class *UMLMethod*, for example, belongs to this chain. This supports the conclusions made by examining the sequence diagrams: when the *modify* button is pressed, a new method is created that presumably replaces the old one. By running another script, the node that represents method *addToParam(UMLParam)* of class *UMLMethod* can be found. It is placed in the bottom right corner of the graph. It is worth noticing that its name has been hidden (i.e. it does not belong to the found chain of nodes). This means that the *addToParam(UMLParam)* method of class *UMLMethod* has not been called, i.e. no parameter has been added for methods, after the *modify* button was pressed. Moreover, it cannot be called under any circumstances from method *modifyButton_actionPerformed(ActionEvent)* with the current implem entation. Our assumption about the source of the bug turned out to be correct—when the *modify* button is depressed the old method is replaced by a new one but the parameters are not copied.

In Shimba, dynamic event trace information can be attached to a static dependency graph. For example, the following edges are given weight values, indicating how many times they have actually been used during the execution: *access*, *assign*, and *call*. In addition, some nodes and edges are usually added to the graph. Such nodes are typically *Exception* nodes, generated when an exception is thrown, and *throw* edges indicating which member function threw the exception. The *throw* edges are naturally given weight values as well. The weight values are stored as arc attribute values in the Rigi graph. The Rigi graph can then be filtered in various ways based on the dynamic information merged into it. For example, parts of the subject system that have not been heavily used can be filtered from the graph by running a script on the graph.

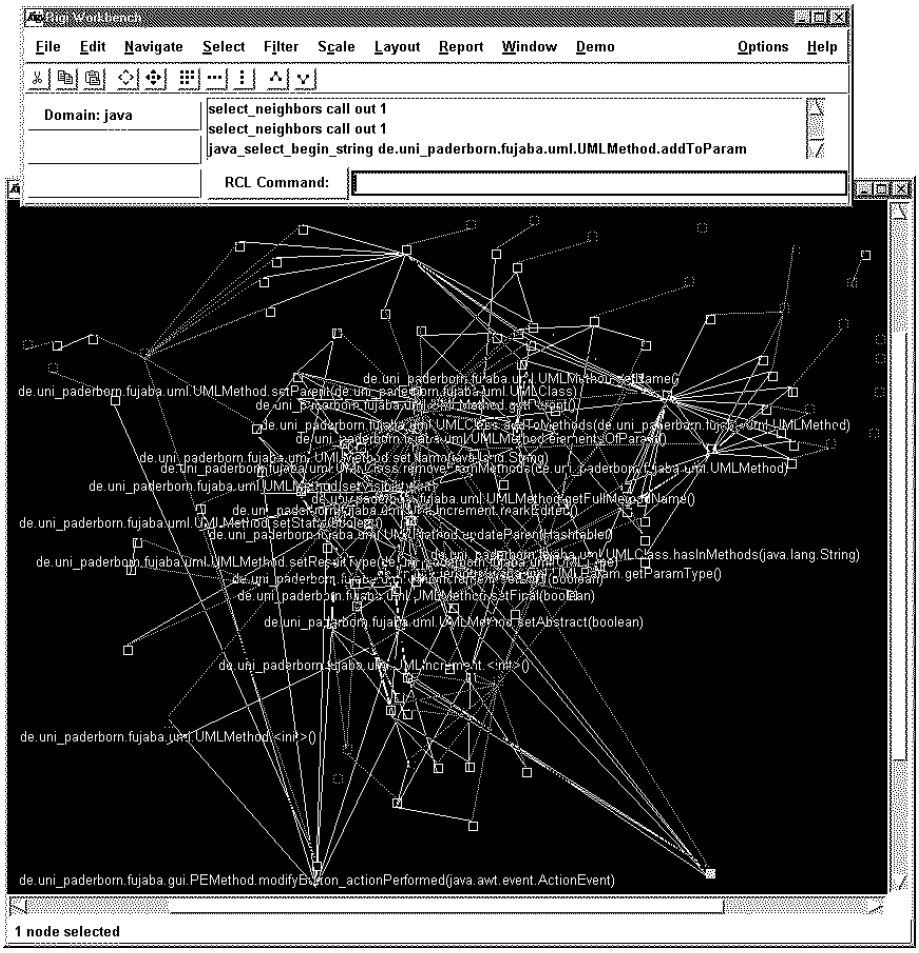

Figure 10. The graph resulting when the initial Rigi graph for FUJABA has been sliced by a set of sequence diagrams.

BUILDING HIGH-LEVEL SEQUENCE DIAGRAMS USING STATIC ABSTRACTIONS

In a previous section, we discussed a method for structuring sequence diagrams using behavioral patterns. In some cases it is also useful to build abstractions for the sequence diagrams based on static criteria. The original sequence diagrams show the interaction among objects or classes. From those sequence diagrams it is difficult to obtain a good impression of the overall communication (e.g., the interaction among subsystems or the interaction between the system and outside participants).

In order to show high-level information with sequence diagrams *horizontal* and *vertical abstractions* can be built. Horizontal abstractions decrease the number of vertical lines in a sequence diagram. For instance, a lower-level sequence diagram might have objects as vertical lines, while in a higher level sequence diagram the vertical lines could represent classes. In that case all the vertical lines that represent instances of a certain class are grouped together and replaced by a single vertical line representing the class itself. A vertical abstraction decreases the number of horizontal arcs in a sequence diagram. They can be built, for example, by omitting 'internal' messages that are sent and received by the same participant or by collapsing method call chains into a single call message. Such approaches are used in Scene [12] and Ovation [14], in which messages can be hidden and expanded with a single mouse click.

In Shimba, both horizontal and vertical abstractions can be constructed. The static abstractions formed in Rigi define the vertical lines that are to be grouped together. When using a reverse engineering tool such as Rigi for defining static abstractions, meaningful groups can be found. Consider a SCED sequence diagram participant C_S that has a corresponding representative C_R in the initial Rigi graph. If C_R has been collapsed into a high-level node, say A_R, then C_S is replaced by a new participant A_S that has the same label as A_R. When constructing high-level sequence diagrams in Shimba, the highest level node is chosen. All the other participants for which A_R is the high-level representative are merged with A_S. Participants for which more abstract representatives cannot be found remain unchanged. The messages in sequence diagrams are changed to identify the owner class or object of the called method.

In a GUI, for example, static abstractions can be constructed to differentiate the view components from the model components (and possibly from the controller components). Using such a high-level static model to raise the level of abstraction of the sequence diagrams, interactions concerning the *Model View Controller (MVC)* structure can be examined. This was also done in the FUJABA case study. In a well constructed Java software system, classes that implement a subsystem are placed in one package (that possibly has subpackages). In Shimba, high-level static views that show dependencies among different packages can be quickly constructed by running a few Rigi scripts. The sequence diagrams can then be modified to visualize the interactions among the packages.

EVALUATION

The Shimba reverse engineering environment performed well in the FUJABA case study. In particular, all the program comprehension and reverse engineering tasks we set out to do for this case study were readily completed with the support of Shimba. The knowledge gained from the interaction between the static and dynamic information at different levels of abstraction was useful and sometimes surprised us. For example, when merging dynamic information into a static view an unexpected installation problem was encountered. There were cases in which the generated information was not particularly useful.

Searching for the behavioral patterns and structuring the SCED sequence diagram with them was one of the most problematic tasks. The string matching algorithms were able to find several (even nested) patterns. When structuring the original sequence diagrams with them, in some cases the readability of the diagrams clearly decreased. This was due to the names and contents of subscenario boxes. Each pattern that is represented as a subscenario is given a name *subsc_x.sc*, where x is the consecutive

number of the subscenario box. Such subscenario boxes do not help the user to get an overview of the execution trace unless named in a more descriptive way. Renaming a subscenario box requires knowledge of its contents (and semantics) and, thus, must be done manually. Furthermore, a pattern contains an arbitrary sequence of SCED sequence diagram elements, formed on the basis of the length of the pattern. Thus, one subscenario might contain SCED sequence diagram elements that do not form a logical unit with its own aims and characterizations. SCED supports navigation through the subscenarios, which helps the user to overcome some of these problems.

Shimba supports both static and dynamic reverse engineering. The case study showed that it is useful to perform static analysis before dynamic analysis. However, Shimba does not require this order. Accomplishing most of the dynamic reverse engineering tasks set for the case study started by selecting a specific part from the Rigi dependency graph. The dynamic information was then generated for the software artifacts that were visualized in that part. In other words, the static information was used to guide the generation of dynamic information. This technique appeared to be very useful for all the goal-driven dynamic reverse engineering tasks. Since run-time information was generated only for a specific part, the debugging was reasonably fast (i.e. the usage of the debugger was only slightly noticeable when running FUJABA). In addition, using Rigi scripts the engineer can quickly and easily find the part of interest and the neighboring parts (i.e. parts that might be communicating with the part in question). Note that she does not need to know the neighboring parts or their names beforehand. Finally, analyzing the SCED sequence diagrams is easy in this case, because uninteresting information is not shown.

The statechart diagram synthesis facility of SCED, combined with the statechart diagram optimization technique, was very useful and practical for analyzing the total behavior of selected objects and methods, even when the resulting statechart diagram was large. The statechart diagram optimization algorithms were applied in most cases since they reduce the size of the statechart diagram significantly. To the best of our knowledge, such features are not provided by other dynamic reverse engineering tools. The statechart diagram allows the engineer to analyze the total run-time usage of an object or a method in a single diagram, disconnected from the rest of the system. Furthermore, it is a powerful and natural graphical representation to examine dynamic control flows. 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.

Shimba supports various debugging strategies. Information about thrown exceptions is essential for understanding the behavior of a target Java software system. It is especially important when the behavior of the target software is unexpected. By adding this kind of information to the SCED sequence diagrams, the engineer can study which exceptions were thrown and by which objects, when they were thrown, and what happened before and after the exceptions were thrown. The SCED sequence diagrams help the engineer to identify exceptional behavior even if the error does not generate any exceptions. Furthermore, the Rigi graph can be sliced by the SCED sequence diagrams to visualize the structure that caused the exceptional behavior. The model slicing technique is also useful in other cases. During the case study, this technique was used a few times to find out why a certain behavior occurs, how parts of FUJABA that were involved in certain sequence diagrams are related to the rest of the system, and what is the underlying structure that causes this behavior. The model slicing technique helps the engineer to understand the context of the sequential behavior.

Attaching dynamic information to a static dependency graph supports both static and dynamic analysis of the target software system. The dynamic information can be used to find heavily used parts of the software and parts that are not used at all. Such information can be used to profile the software. However, to understand the behavior of the software fully, sequential information is needed.

In Shimba, the level of abstraction of the SCED sequence diagrams can be raised using static abstractions constructed using Rigi. This technique helps the engineer to get an overall picture of the behavior. Rigi allows the engineer to make arbitrary static abstractions. This makes the sequence diagram abstraction technique especially useful. For example, the technique can be used to ensure that the static abstractions are meaningful and to understand how different high-level components communicate with each other. In this case study, the technique was used a few times for these purposes.

In practice, combining static and dynamic reverse engineering was very useful. All the reverse engineering techniques of Shimba were used several times during the case study. Using Shimba, the engineer can understand how static and dynamic views are connected with each other, which is one of the most difficult tasks in reverse engineering object-oriented software systems. The static views provide the context for the dynamic analysis, and the dynamic views show what the current structure of the software means in practice.

The techniques supported by Shimba are also useful for forward engineering. The engineer can use Rigi to view the current structure of the software and check if the design guidelines have been followed. The current behavior can be examined with the SCED sequence and statechart diagrams and checked against use case specifications.

RELATED RESEARCH

Many tools supporting forward engineering of object-oriented software are also able to extract class diagrams for existing software. Tools for viewing the run-time behavior of a target software system often use variations of MSCs as a visualization technique. Such tools include Jinsight [10] and Scene [12]. In this section, we relate our approach to tools, environments, and methods that aim at combining static and dynamic information.

ISVis [11] is a visualization tool that supports the browsing and analysis of execution scenarios. A source code instrumentation technique is used to produce the execution scenarios. The user can restrict the amount of event trace information to be generated by selecting a list of files, for which breakpoints will be set. 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*. The static information about files, classes, and functions belonging to the target software are listed in a *Main View* of ISVis. The view allows the user to build high-level abstractions of such software actors through containment hierarchies and user-defined components. A high-level scenario can be produced based on static abstractions. In Shimba, scripts can be run on a Rigi graph to construct abstractions based on containment or any other relationship. The user can also construct arbitrary abstractions manually. A high-level SCED sequence diagram can then be produced based on these static abstractions. 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. Furthermore, the algorithms are able to find patterns only if they lay entirely inside one SCED sequence diagram.

Program Explorer [13] combines static information with run-time information to produce views that summarize relevant computations of the target system. To reduce the amount of run-time information generated, the user can choose when to start and stop recording events during the execution. In our approach, the user can decide how the event trace is split into sequence diagrams and examine only those of interest. Moreover, Rigi is used to reduce the amount of event trace information generated. Merging, pruning, and slicing techniques are used in Program Explorer for removing unwanted information from the views. The user can not, however, choose freely the level of abstraction on which she wants to view and examine the information. The granularity of components viewed can not be greater than a single class.

Walker *et al.* use high-level models for visualizing program execution information [24]. 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. Histograms and annotations can be attached to the boxes and edges. A *cel view* shows events that occurred within a particular interval of the system's execution. Summary views can be used to examine all events occurring in the trace. The system is written in Smalltalk and is used to analyze Smalltalk programs. 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, static and dynamic information is shown in separate views and the high-level static components are constructed using Rigi. The user can then build high-level sequence diagrams using a mapping between low-level software artifacts and high-level components.

Sefika *et al.* introduce an architecture-oriented visualization approach that can be used to view the behavior of a target system at different levels of granularity [25]. They introduce a technique called *architectural-aware instrumentation*, which allows the user to gather information from the target system at the desired level of abstraction. Such levels include subsystem, framework, pattern, class, object, and method levels. The extracted information is shown using appropriate views. The instrumentation mechanism used hard-wires the level of abstraction into the source code instrumentation process, which makes the approach somewhat inflexible. The user has to decide the level of abstraction and views to be generated before running the target software. In our approach, various techniques to build abstractions from the low-level views are provided.

In [26] a query-based approach to recover high-level views of object-oriented applications is presented. Static and dynamic aspects of the target software are modeled in terms of logic facts. By making different queries on the facts, the user can decide what kind of views are to be produced. The views can, for example, contain static and/or dynamic information and model the information on different levels of abstraction. The queries also provide a way to restrict the amount of information generated. This approach does not support direct information exchange among different views.

Dali is a workbench for architectural extraction, manipulation, and conformance testing [27]. It integrates several analysis tools and saves the information extracted in a repository. Dali uses the merged view approach, modeling the information as a Rigi graph. The user can organize and manipulate the view and hence produce other refined views at a desired level of abstraction.

CONCLUSIONS

Program understanding techniques are useful for various software engineering tasks. Software exploration tools are needed for software maintenance, re-engineering, reuse, and forward engineering purposes. Several reverse engineering tools and environments that help the engineer to understand the structure of the software are currently available. Tool support for understanding behavioral aspects of a subject system also exists. However, there are few tools that support both of these tasks and help the engineer to understand the relations between behavioral and structural aspects of a software system. The Shimba environment was developed to help the engineer reach this goal.

The UML notation contains several diagram types that can be used to model a system from different perspectives and at different levels of abstraction. Those diagrams share information and depend on each other in various ways, thus enabling the development of versatile model transformation and synthesis techniques. Such techniques are especially useful for reverse engineering, if the target model represents information at a higher level of abstraction than the source model. For example, the generation of UML component diagrams would be useful for understanding the high-level architecture of the subject object-oriented software system. Furthermore, the use of transformation and synthesis mechanisms is not limited to two diagram types. A more refined model can usually be constructed if the information is gathered from different sources. In addition, information included in a single diagram could be used to refine several existing models. Examining the dependencies among the UML diagrams and exploring new transformation and synthesis algorithms will be part of our future research.

Practical experiences are essential for developing useful reverse engineering techniques and tools. An industrial tool provides an ideal platform to research and test reverse engineering techniques: the usefulness of the techniques can be studied with real examples by the users of the tool. On the other hand, a reverse engineering environment that uses the different UML diagram types would be desirable. A part of our future work is to investigate possibilities to integrate the reverse engineering techniques of Shimba into a real-world UML modeling tool, the Nokia TED tool [28]. In addition, implementing the static and dynamic reverse engineering techniques in one tool, TED, allows for tighter integration compared to existing reverse engineering environments.

ACKNOWLEDGEMENTS

The research has been financially supported by the Academy of Finland, the National Technology Agency of Finland (TEKES), Nokia, Neles Automation, Sensor Software Consulting, Acura Systems, and Plenware.

REFERENCES

1. Systä T. On the relationships between static and dynamic models in reverse engineering Java software. *Proceedings of WCRE 1999*. IEEE Computer Society Press, 1999; 304–313.
2. Rockel I, Heimes F. FUJABA—Homepage. http://www.uni-paderborn.de/fachbereich/AG/schaefer/ag_dt/PG/Fujaba/fujaba.html [2000].
3. Müller H, Wong K, Tilley S. Understanding software systems using reverse engineering technology. *Object-Oriented Technology for Database and Software Systems*, Alagar VS, Missaoui R (ed.). World Scientific, 1995; 240–252.
4. Müller H, Klashinsky K. Rigi—A system for programming-in-the-large. *Proceedings of ICSE 1988*. IEEE Computer Society Press, 1988; 80–86.
5. Sun Microsystems Inc. Java(TM) 2 SDK, Standard Edition Version 1.2. http://java.sun.com/products/jdk/1.2/ [2000].

6. Koskimies K, Männistö T, Systä T, Tuomi J. Automated support for modeling oo software. *IEEE Software* 1998; **15**(1):87–94.
7. Rumbaugh J, Jacobson J, Booch G. *The Unified Modeling Reference Manual*. Addison-Wesley, 1999.
8. Rational Software Corporation. The Unified Modeling Language Notation Guide v.1.3. http://www.rational.com [2000].
9. International telecommunications union—Standardization (ITU-T). *ITU-T Recommendation Z.120: Message Sequence Chart (MSC)*. ITU-T: Geneva, 1996.
10. IBM Research. Jinsight, visualizing the execution of Java programs. http://www.research.ibm.com/jinsight/ [2000].
11. Jerding D, Rugaber S. Using visualization for architectural localization and extraction. *Proceedings of WCRE 1997*. IEEE Computer Society Press, 1997; 56–65.
12. Koskimies K, Mössenböck H. Scene: Using scenario diagrams and active text for illustrating object-oriented programs. *Proceedings of ICSE 1996*. ACM Press, 1996; 366–375.
13. Lange DB, Nakamura Y. Object-oriented program tracing and visualization. *IEEE Computer* 1997; **30**(5):63–70.
14. De Pauw W, Lorenz D, Vlissides J, Wegman M. Execution patterns in object-oriented visualization. *Proceedings of COOTS 1998*. USENIX, 1998; 219–234.
15. Systä T. Understanding the Behavior of Java Programs. *Proceedings of WCRE 2000*. IEEE Computer Society Press, 2000; 214–223.
16. Systä T. Static and dynamic reverse engineering techniques for Java software system. *Ph.D. Dissertation*, University of Tampere, Department of Computer and Information Sciences. Report A-2000-4, 2000.
17. Wong K. Rigi User's Manual. http://www.rigi.csc.uvic.ca [1998].
18. Tilley S, Wong K, Storey M-A, Müller H. Programmable reverse engineering. *International Journal of Software Engineering and Knowledge Engineering* 1994; **4**(4):501–520.
19. Chidamber SR, Kemerer CF. A metrics suite for object-oriented design. *IEEE Transactions Software Engineering* 1994; **20**(6):476–493.
20. Systä T, Yu P, Müller H. Analyzing Java software by combining metrics and program visualization. *Proceedings of CSMR 2000*. IEEE Computer Society, 2000; 199–208.
21. Koskimies K, Mäkinen E. Automatic synthesis of state machines from trace diagrams. *Software—Practice and Experience* 1994; **24**(7):643–658.
22. De Pauw W, Helm R, Kimelman D, Vlissides J. Visualizing the behavior of object-oriented systems. *Proceedings of OOPSLA 1993*. ACM Press, 1993; 326–337.
23. Boyer RS, Moore JS. A fast string searching algorithm. *Communication of the ACM* 1997; **20**(10):762–772.
24. Walker R, Murphy G, Freeman-Benson B, Wright D, Swanson D, Isaak J. Visualizing dynamic software system information through high-level models. *Proceedings of OOPSLA 1998*. ACM Press, 1998; 271–283.
25. Sefika M, Sane A, Campbell RH. Architecture-oriented visualization. *Proceedings of OOPSLA 1996*. ACM Press, 1996; 389–405.
26. Richner T, Ducasse S. Recovering high-level views of object-oriented applications form static and dynamic information. *Proceedings of ICSM 1999*. IEEE Computer Society Press, 1999; 13–22.
27. Kazman R, Carriere J. Playing detective: Reconstructing software architecture from available evidence. *Journal of Automated Software Engineering* 1999; **6**(2):107–138.
28. Wikman J. Evolution of a distributed repository-based architecture. http://www.ide.hk-r.se/~bosch/NOSA98/JohanWikman.pdf [1998].

