On the Relationships between Static and Dynamic Models in Reverse Engineering Java Software

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

Abstract

An experimental environment for reverse engineering Java software is discussed. Static information is extracted from class files and viewed using Rigi reverse engineering environment. The dynamic information is generated by running the target software under a debugger. The debugged event trace information is viewed as scenario diagrams using a prototype tool called SCED. In SCED state diagrams can be synthesized automatically from scenario diagrams. Dynamic information can also be attached to the static Rigi graph. Both static and dynamic views contain information about software artifacts and their relations. Such overlapping information forms a connection for information exchange between the views. SCED scenario diagrams are used for slicing the Rigi view and the Rigi view, in turn, is used to guide the generation of SCED scenario diagrams and for raising their level of abstraction.

1. Introduction

The change of programming languages and styles drives changes in current reverse engineering tools and methods. Today’s legacy systems are written in COBOL or C, while tomorrow’s legacy systems are written in C++ and Java. The adoption of object-oriented programming paradigm has changed programming styles dramatically. The dynamic nature of object-oriented programs emphasizes the importance of dynamic reverse engineering. Object creation, object deletion/garbage collection, and dynamic binding make it very difficult, and most cases impossible, to understand the behavior by just examining the source code.

One of the most challenging tasks in reverse engineering is to build descriptive and readable views of the software on the right level of abstraction. The extracted information can be merged into a single view, and support for slicing techniques and means to build abstractions can be provided in order to keep the view readable and understandable. However, when both static information and dynamic information are considered, the chosen view often serves either the static or the dynamic aspect but rarely both. In practice, the dynamic information is just viewed against a previously built static model. Another approach is to distinguish dynamic models from static ones. Since this is done in forward engineering, it is natural to do so also in reverse engineering. Like in forward engineering, having separate views requires a meaningful and consistent connection between them.

An experimental environment has been built to reverse engineer Java software. The static information is extracted from Java byte code. It can be saved in Rigi Standard Format (RSF) and viewed with the Rigi reverse engineering tool [10]. The dynamic event trace information is generated automatically as a result of running the target system under a debugger. The event trace can be saved in the SCED scenario diagram format and viewed with a prototype tool SCED [6]. In SCED state diagrams can be synthesized automatically from scenario diagrams. This facility is used to examine the total behavior of an object or a single method, disconnected from the rest of the system.

In this paper examples of information exchange between static and dynamic views are discussed. The overlapping information of the views can be used as a connection that enables the information exchange between the views. The views can thus be used to comprehend each other. A target Java software has been examined to clarify the methods discussed. The selected target system, FUJABA [14], is freely available software and was developed at the University of Paderborn. The primary topic of the FUJABA project and
environment is Round Trip Engineering with UML, SDM (Story Driven Modeling), Java and Design Patterns. FUJABA provides editors for defining both structural and behavioral aspects of a software. FUJABA is written in Java, and it contains almost 700 classes. The FUJABA version under examination was 0.6.3-0. The focus during the reverse engineering process was on studying the structure and functionality of the class diagram editor of FUJABA.

2. Viewing the extracted information

Static information of the software consists of software artifacts and their relations. In Java, for example, such artifacts could be classes, interfaces, methods, and variables. The relations might include extension relationships between classes or interfaces, method calls between methods, and so on. Dynamic information contains software artifacts as well. In addition, it contains sequential event trace information, information about concurrency, memory management and leaks, code coverage, etc.

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

2.1. Using a single view

Merging dynamic and static information into a single view has both advantages and disadvantages. A single view would directly illustrate connections between static and dynamic information. In addition, the quality of the view can be improved and insured when merging static and dynamic information. For example, if the static information is extracted from the source code some of the artifacts and relations might have been ignored. Such an artifact is, for instance, a default constructor that is not explicitly written in the source code. Also calls of such default constructors are ignored. Dynamic analysis of the software produces these pieces of information since the default constructors are actually invoked when an instance of a class is created. In some cases, however, static information might be confusing when compared to the dynamic information. For instance, because of polymorphism, a method call written in the source code represents a set of possible operations, rather than a method call that is actually invoked at runtime.

Building abstractions for merged views can be difficult because static and dynamic abstractions usually differ considerably. Dynamic abstractions are typically behavioral patterns or use cases, whereas static abstractions are subsystems. For example, most of the classes used by two use cases “withdrawing money using an ATM” and “paying a bill using an ATM” are the same and may belong to a single subsystem “ATM”. The user therefore has to choose whether to build the abstractions from static or dynamic point of view at a very early stage.

Forming merged views themselves might be complicated. For example, it is easy to add code coverage information to a static view but it is much more difficult to add information about concurrency or sequential behavior to it. In UML, collaboration diagrams can be used to view both static and dynamic event trace information. However, even moderate size collaboration diagrams easily become hard to read and in reverse engineering the amount of extracted information is typically very large. In general, the more information attached to a single view, the less readable it becomes, thus losing one of its main purposes. Slicing techniques can be used to filter out uninteresting information and to focus on desired aspects of the software. On the other hand, if slicing provides the only means to focus on the chosen aspect of the software, e.g., sequential event trace information, then merging that information to the view is questionable. Unless the merge serves another purpose, choosing more suitable and descriptive view would probably promote better the reverse engineering task.

2.2. Using a set of different views

In forward engineering different diagrams are used to model the static structure and dynamic behavior of the software. In UML there are static diagrams, dynamic diagrams, and diagrams that model both static and dynamic aspects of the software. From a large set of diagrams, the user chooses the ones that best suit her purposes. Ideally, this should be the case also in reverse engineering. If a large set of diagrams is chosen, the problem of keeping them consistent and connected to each other needs to be considered. On the other hand, a single diagram is often insufficient to model the software and the problems explained in Section 2.1 occur. The number and type of diagrams to be used depend on the purpose and needs in the same way as in forward engineering.

Distinguishing static and dynamic views allows showing information that would be hard, or even impossible, to include in a single merged view. This, in turn, offers extended possibilities to support program slicing, requiring that there is a connection that enables information exchange
between the views. For example, if scenario diagrams are used for viewing the event trace information, the static model can be sliced based on the information included in a desired set of scenarios. The resulting slice shows the structure of a part of the software that causes that behavior. Furthermore, the static knowledge of the software can be used to guide the generation of dynamic information, i.e., to focus on the behavior of the desired parts of the software.

When using a set of different views abstractions for dynamic views can be built according to different principles than the abstractions for static ones. For example, behavioral patterns can be used to raise the level of abstraction of scenario diagrams, while structural dependencies can be used as a criteria when building abstractions to static views. Forcing the dynamic information to be abstracted based on static criteria would probably hide some essential features in the behavior and make it more complicated to understand the overall behavior. However, in some cases it might be meaningful, e.g., to modify scenario diagrams to show interaction between high level static components instead of showing the interaction between classes or even objects.

3. Experimental solution for visualizing the extracted information

In this experiment Rigi is used for viewing and analyzing the static structure of the software extracted from Java byte code. SCED is used for modeling the event trace information generated when running the target system under a customized jdk debugger. The debugger is implemented using public classes of sun.tools.debug package of jdk1.2. Dynamic code coverage information can be attached to the static Rigi view when debugging a target software. The constructed views can then be used to comprehend each other as discussed in Sections 4, 5, and 6. Figure 1 shows the overall structure of the system.

3.1. Constructing static dependency graph

The software artifacts and their dependencies are extracted from Java class files. The extracted information includes the following Java software artifacts: classes, interfaces, methods, constructors, variables, and static initialization blocks. The extracted dependencies between 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 assign relationship (e.g., a method assigns a value for a variable), and so on. The implementation of the extractor is written in Java. It uses some of the public classes of sun.tools.java package of jdk1.2. The static information is extracted following the descriptions of the contents of Java class files [17].

In Rigi all the software artifacts are shown as nodes, and relationships as directed edges between them. The type of a node or an edge is identified by a specific color. Extracting the static information of FUJABA software using the byte code extractor resulted in a static dependency graph with 25854 nodes.

3.2. Constructing dynamic models

The user can select the classes and/or a set of methods for which the dynamic event trace information is collected. The debugger can be given instructions to put breakpoints for all or selected methods (entries) of a desired set of classes. The user can also choose exceptions to be tracked. Event trace information is generated whenever a breakpoint is hit or an exception is thrown. If the thrown exception is among the set of exceptions selected by the user, then the class of that exception is set to be the receiver for the event. Otherwise, the receiver will be the superclass of all exceptions, namely class java.lang.Exception. If the user has not selected any exceptions, java.lang.Exception class will be the receiver for all exception events. For events that represent method, constructor, and static block invocations, the receivers and senders can be set to be either classes or objects. When loading the event trace information to SCED receivers and senders of events are represented as scenario participants, and shown as vertical lines. Furthermore, the event trace is split into several scenarios in order to limit
the size of a single scenario. If not guided by the user, this is done automatically after the event trace has grown big enough.

In [5], it has been demonstrated how a minimal state machine, which is able to execute all the given scenarios with respect to a certain object, can be synthesized automatically. This algorithm with a few modifications has been implemented in SCED. When synthesizing a state diagram for a participant, each scenario diagram is traversed from top to bottom from the point of view of that participant. Each received event is mapped to a transition in the state diagram. Sent events are regarded as primitive actions that are associated with states. A synthesized state diagram can be simplified while preserving its information content by generating advanced state diagram concepts for it. For example, the simplified state diagram allows several actions to be placed into a single state, actions attached to transitions, entry and exit actions of states, etc.

By using a single menu command the user can select one participant in a scenario and synthesize a state diagram automatically for it. 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. In that case, the user selects a method call event. The scenario is again traversed from top to bottom but only between the method call event and a corresponding return event. The synthesis algorithm is described more detailed in [5, 16], and the state diagram simplification algorithms are discussed in [16]. Relationships between static Rigi views and synthesized state diagrams are not discussed in this paper.

Dynamic event trace information can also be attached to a static dependency graph. When merging dynamic information to the static view, access, assign, and call edges are given weight values indicating how many times they have actually been used during the execution. 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 method, constructor, or static block threw the exception. The throw edges are given weight values as well. Figure 2 shows a Rigi graph for which dynamic information has been merged. A small dialog has opened to show attribute values of a throw edge entering java.lang.MissingResourceException node. The weight value indicates that the corresponding exception was thrown 221 times. The type of the exception and the number of times it was thrown refers to an installation problem.

**Figure 2.** A Rigi view containing both static and dynamic information. The opened dialog shows that java.lang.MissingResourceException has been thrown 221 times at run-time.

4. Using Rigi views to guide the generation of dynamic information

When reverse engineering an unknown system, the engineer might be interested in a specific part of the software. If she wants to study the dynamic behavior of the part in question, 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. Such information slicing can dramatically decrease the size of generated event traces, still containing the information needed. In order to be able to do that, the engineer has to know how classes depend on each other, i.e., the static structure of the software.

The static model can then be used to define the desired part of the software and all its dependencies. In Rigi such parts can easily be defined. Assume that the user is interested in the behavior of a set of classes. It is advantageous then to filter out everything else from the Rigi graph except a set of nodes representing those classes, a set of nodes representing members of those classes, and all neighbor nodes of these two sets. The user may still want to filter out some of the remaining nodes. For example, she might find nodes that represent software artifacts outside the target system uninteresting.

Assume that the user is interested in the behavior
of the dialog in Figure 4. By running a simple script on the initial graph representing the whole FUJABA system the interesting parts of the software can be easily determined. Figure 3 shows a Rigi graph that contains only the nodes that represent the dialog class \texttt{de.uni.paderborn.fujaba.gui.P EMethod} itself, all its members, and all their neighbor nodes. The rest of the graph has been filtered out. Also nodes representing software artifacts that do not belong to FUJABA itself, e.g., jdk or Swing classes and methods, have been filtered out. This has been done assuming that the behavior of such classes does not considerably help understanding the behavior of the target system itself and does not have a crucial effect on the behavior of the target system. In addition, tracking down the invocations of methods belonging to packages outside the target system might increase the size of the event trace dramatically, containing more or less useless information. For example, methods of class \texttt{java.lang.String} are frequently called but examining such calls is not necessary for understanding the behavior of the dialog. Nodes that represent methods, constructors, and static blocks are selected and their names can be given to the debugger for breakpointing. In this case breakpoints will be put on the first line of 89 methods or constructors. The whole FUJABA software contains altogether 7151 methods, constructors, or static blocks. Breakpointing all of them would require a lot of memory and debugging the software would be very slow.

Restricting the generation of event trace information is a powerful way to avoid event explosion. It is natural to do that based on the static structure of the program. This approach enables the user to focus on the specific part of the software. Using a proper static reverse engineering tool is essential for taking full benefit of the approach, mainly by giving support for safety, consistency, and efficiency.

5. Slicing Rigi views using SCED scenarios

Section 3.2 discussed how dynamic information can be attached to a static Rigi graph. The Rigi graph can then be sliced based on this dynamic information. For example, parts of the software that have not been heavily used can be filtered from the graph by running a script on the graph. If the script takes a parameter as a threshold value for weight attributes of arcs, it can be used for defining the most heavily used parts of the software. A slightly different approach is taken when a purely static Rigi graph is sliced using SCED scenarios.

By examining the SCED scenarios a called method can be identified but the caller is given as an object or a class. However, the method is always called from another method, constructor, or a static block. The static Rigi graph also shows a \textit{call} dependence between two methods rather than between a calling class and a called method. Slicing the purely static Rigi graph based on example scenarios hence differs from slicing a Rigi graph, in which dynamic information has been merged. In the former case 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 slicing algorithms can be run from Rigi.

The dynamic slicing approach can be used for studying the parts of the target software that are involved in a specific kind of usage. For example, a bug in the software might cause a failure in the execution inevitably or only in certain cases. By examining the parts of the software that are involved in a usage that causes a failure the user might be able to conclude whether the failure was inevitable or not.

In FUJABA methods can be edited and created using a dialog shown in Figure 4. There is a known bug in the functionality of the dialog: if the user selects a method, for which parameters have been given, and presses the \textit{modify} button, the parameters disappear. This bug is listed in the
For tracing down the source of the bug, FUJABA was run under the debugger. Breakpoints were set for all methods of following classes: UMLClass, UMLMethod, UMLParam, and the dialog class PEMethod itself. The usage for generating the run-time information included opening the dialog, giving a new parameter for a selected method by using a dialog that is opened when parameter button is pressed, and pressing modify button. By studying the resulting scenarios it can be seen that a method addToParam(UMLParam) of class UMLMethod is called by class PEParameters (a class that implements the dialog used for creating and editing parameters) when parameters were added for a method. This can be concluded from the scenario in Figure 5. Figure 6 shows one of the later scenarios. The constructor invocation <init>() of class de.uni.paderborn.fujaba.uml.UMLMethod in it indicates that when the modify button is pressed, i.e., modifyButton.actionPerformed(ActionEvent) method is called, a new method is created. This hints that the old method is replaced by a new one.

For examining the structure of the parts of FUJABA that might contain the source of the bug, the initial Rigi graph representing the whole FUJABA software was sliced based on a set of scenarios that were generated after the dialog in Figure 4 was opened. There were altogether 29 such scenarios. The resulting Rigi graph is shown in Figure 7. The node representing method modifyButton.actionPerformed(ActionEvent) of class PEMethod can be seen in the bottom of the graph. The nodes, names of which have not been hidden, 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 script on the graph. The default constructor of class UMLMethod, e.g., belongs to this chain. This supports the conclusions made by examining the scenarios: when the modify button is pressed, a new method is created that presumably replaces old one. By running another script 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 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(java.awt.event.ActionEvent) with the current implementation. Our assumption about the source of the bug emerged to be true: when the modify button was pressed the old method was replaced by a new one but the parameters were not copied.
6. Building high-level scenarios using static abstractions

Even a brief usage of the target system typically results in many SCED scenarios during the dynamic reverse engineering process. Hence, means to raise the level of abstraction of the scenarios are needed. A commonly used approach is to search for behavioral patterns. This approach is used in Jinsight [12] and Ovation [11]. In some cases it is meaningful to build abstractions for the scenarios based on static criteria. The original scenarios show the interaction between objects or classes. From those scenarios it is difficult to get a flavor of the overall communication, e.g., the interaction between packages or the interaction between the system and outside participants.

In order to show high level information with scenarios horizontal and vertical abstractions can be built. Horizontal abstractions decrease the number of vertical lines in a scenario. For instance, a lower level scenario might have objects as vertical lines, while in a higher level scenario 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. Vertical abstraction decreases the number of horizontal arcs in a scenario. They can be built, e.g., by omitting “internal” events that are sent and received by the same participant or by collapsing method call chains into a single call event. Such approaches are used in Scene [7] and Ovation [11], in which events can be hidden and expanded by a simple mouse click.

The approach to be introduced next can be used to build both horizontal and vertical abstractions. The static abstractions formed in Rigi define the vertical lines that will be grouped together. When using a reverse engineering tool for defining static abstractions, meaningful groups can be found. Consider a SCED scenario 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 $A_R$. Because nodes can be nested to an arbitrary depth in Rigi, 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 representative cannot be found remain unchanged. The events in scenarios are changed to identify the owner class or object of the called method.

Vertical abstractions are made automatically when internal method calls occur. In SCED scenarios rectangular action boxes are used for showing internal events, i.e., events for which the sender and the receiver are the same. All such action boxes are removed from the changed scenarios as well as events that would be represented as action boxes after the conversion. This shortens the scenarios and emphasizes the interaction between the high-level participants.

In FUJABA, classes that represent UML notation concepts belong to package `de.uni.paderborn.fujaba.uml`. The editors and dialogs used for defining and editing these concepts are defined in `de.uni.paderborn.fujaba.gui` package. For example, the dialog in Figure 4 is implemented as class `de.uni.paderborn.fujaba.gui.PEMethod`. 

---

**Figure 6.** Pressing the modify button causes a constructor invocation of class UMLMethod.

**Figure 7.** The graph resulting when the initial Rigi graph for FUJABA software has been sliced by a set of scenarios.
and it is used for defining and editing methods that are represented as `de.uni.paderborn.fujaba.uml.UMLMethod` objects. Altogether 24 editor classes have a superclass `PropertyEditor`, from which the individual editors are derived. Similarly, class `UMLIncrement` is a superclass for 46 UML notation concepts.

For capturing high-level information about the model-view structure of FUJABA class diagram editor few static abstractions were made. First, class `PropertyEditor` and all its subclasses were collapsed into a single `PropertyEditor2` node. Second, class `UMLIncrement` and all its subclasses were collapsed into a single `UMLIncrement2` node. Third, all classes that implement jdk’s `ActionListener` interface were collapsed into a single `ActionListener2` node. There were altogether 92 such classes. Those classes represent GUI items that are able to receive action events, e.g., user inputs. When reverse engineering the structure and behavior of FUJABA class diagram editor these classes represent the system border. Forth, class `PESelection` and all its subclasses were collapsed into a single `PESelection2` node. Those classes are used by several dialogs in FUJABA. They provide the user a list to which she can add items using an ‘add’ button and from which she can remove items using a ‘remove’ button.

A set of SCED scenarios were generated when reverse engineering the dynamic interaction between the model and view components. Breakpoints were set for all methods of the following classes: `UMLMethod`, `UMLParam`, `UMLClass`, `UMLAttr`, `PEMethod`, `PEParameter`, `PEClass`, and `PEVariable`. The usage included adding and editing classes, methods, variables, and parameters. The resulting scenarios were (automatically) modified using the high-level Rigi components. The number of scenarios was decreased from 26 to 7 and the number of scenario items was decreased from 2038 to 513. Figure 8 shows a corner of one of the resulting scenarios. The scenario models usage in which a parameter button of the dialog in Figure 4 was pressed. That caused a dialog for defining and editing parameters to be opened. Using that dialog a new parameter was added and modified. The usage is clearly recognizable from the scenario. Three events corresponding to user inputs are selected. Furthermore, participants representing the system border (`javax.swing.AbstractButton` and `java.awt.event.ActionEvent`) and the view components (`de.uni.paderborn.gui.PropertyEditor2` and `de.uni.paderborn.gui.PESelection2`), and the model component (`de.uni.paderborn.uml.UMLIncrement2`) can be easily distinguished. The high-level scenario thus clarifies the user interaction as well as the communication between the view and the corresponding model components.

7. Related research

Several tools and environments supporting reverse engineering and architecture recovery rely on static analysis of the software, for example, Rigi [10], Bookshelf [2], CIAO [1], Sniff+, etc. In addition, 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 often use variations of Message Sequence Charts (MSCs) as a visualization technique. Such tools include Jinsight [12] and Scene [7]. Methods and tools that allow viewing the run-time behavior on a desired level of abstraction are introduced, for instance, in [18] and [15]. Here our approach is related to tools, environments, and methods that aim at combining static and dynamic information.

ISVis [3] is visualization tool that supports the browsing and analysis of execution scenarios. 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 a 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. A corresponding method in our approach is described in Section 6. 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. The precise description of the method is out of the scope of this paper.

Program Explorer [8, 9] 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 scenarios 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 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 to examine the information. The granularity on components viewed can not be greater than a single class.

In [13] 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 will 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 between different views.

Dali is a workbench for architectural extraction, manipulation, and conformance testing [4]. 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 on a desired level of abstraction.

8. Discussion

Reverse engineering aims at analyzing and modeling the target software so that it is easier to understand, maintain, re-engineer, and reuse. This goal can be achieved by producing design models from the software. To fully understand existing software both static and dynamic information need to be extracted. Static information describes the structure of the software the way it is written in the source code, while dynamic information describes its run-time behavior. Extracting information about the dynamic behavior of the software is especially important when examining object-oriented software. This is due to the dynamic nature of object-oriented programs: object creation, object deletion/garbage collection, and dynamic binding make it very difficult, and in most cases impossible, to understand the behavior by just studying the source code.

In this paper reverse engineering Java software is discussed. Both static and dynamic models are built to help the user to analyze different aspects of the software. Static information is extracted from class files and viewed using a reverse engineering environment Rigi. The dynamic event trace information is generated by running the target software under a debugger and is viewed as scenario diagrams using a dynamic modeling tool SCED. In SCED state diagrams can be synthesized automatically from scenario diagrams. This facility can be used for examining the overall behavior of a selected object or a method, disconnected from the rest of the system. Both static and dynamic analysis contain information about software artifacts and their relations. This can be used as a connection that enables information exchange between the models. The models are then used to comprehend each other and for providing extended support for program slicing. In this paper we have not used the state diagram synthesis facility of SCED. However, it provides obvious possibilities to support understanding the behavior of high-level static components.

Most currently available reverse engineering tools focus on either static or dynamic aspect of the software but rarely on both. In forward engineering OOAD methodologies provide different models that can be used in analysis and design phases to model the static structure and dynamic behavior of the software. Some of them contain strictly static information, some are used for dynamic modeling, and some model both static and dynamic aspects of the software. It would be natural to do so also in reverse engineering. Moreover, it is desirable that the reverse engineering tool is able to produce similar diagrams as
are used in forward engineering. Combining reverse engineering and forward engineering tools that use similar models would give more support for re-engineering and round-trip-engineering.

The application of reverse engineering techniques is not limited to understanding old legacy systems. They can and should be applied for forward engineering purposes as well. In software development reverse engineering the current static structure of the software helps the engineer to insure that the architectural guidelines are followed, to get an overall picture of the software, to document the implementation steps, and so on. Reverse engineering the run-time behavior during the software development phase is essential for profiling, tracing the sources of bugs, understanding and insuring the current behavior of the software, finding dead code, etc. Applying reverse engineering techniques during the software development phase also supports documentation, hence helping us to avoid similar problems with our Java code, as we currently have to face with COBOL and C code.

Acknowledgements

My research has been funded by the Academy of Finland and the Tampere Graduate School in Information Science and Engineering (TISE). I would like to express my gratitude to Kai Koskimies for supervising my research. A great part of the research was done while visiting the University of Victoria. I would like to thank Hausi Müller for that opportunity. I would also like to thank Erkki Mäkinen and the anonymous reviewers for their valuable feedback.

References