Static and Dynamic Reverse Engineering Techniques for Java Software Systems

Acta Electronica Universitatis Tamperensis 30

TARJA SYSTÄ

Static and Dynamic Reverse Engineering Techniques for Java Software Systems

University of Tampere
Tampere 2000

ACADEMIC DISSERTATION
University of Tampere, Department of Computer and Information Sciences
Finland

Acta Electronica Universitatis Tamperensis 30
ISBN 951-44-4811-1
ISSN 1456-954X
http://acta.uta.fi

TARJA SYSTÄ

Static and Dynamic Reverse Engineering Techniques for Java Software Systems

■

ACADEMIC DISSERTATION

To be presented, with the permission of
the Faculty of Economics and Administration
of the University of Tampere, for public discussion
in the Paavo Koli Auditorium of the University,
Kehruukoulunkatu 1, Tampere, on May 8th, 2000 at 12 o'clock.

University of Tampere
Tampere 2000

Acknowledgements

I am very grateful to my supervisor Kai Koskimies for all his support. Over the years, Kai has encouraged me through my Licentiate and PhD studies. He has given me a lot of feedback and many useful pieces of advice, every time I needed them. I would also like to thank Erkki Mäkinen for proofreading my papers, encouraging and guiding me in my studies, and being always able to find answers for all kinds of questions. Kai hired me in 1993 as a researcher for the SCED research project for almost three years. It was a pleasure and privilege to work with Jyrki Tuomi and Tatu Männistö on SCED. The SCED project was financially supported by the Center for Technological Development in Finland (TEKES), Nokia Research Center, Valmet Automation, Stonesoft, Kone, and Prosa Software.

After the SCED project, my PhD studies have been financially supported by Tampere Graduate School in Information Science and Engineering (TISE). The funding I received from TISE allowed me to fully concentrate on my PhD studies and to visit the University of Victoria, Canada, during years 1997-1998. The visit was partly funded by the Academy of Finland. I am grateful to Hausi Müller for welcoming me to the Rigi research project at UVic. He gave me a good opportunity to continue my studies, and made it easy and pleasant for me to work and collaborate with the Rigi members. I enjoyed those one and half years I was able to spend in Victoria.

I would like to express my gratitude to the reviewers of the dissertation, Hausi Müller and Jukka Paakki. Their feedback was useful for improving the work. I would also like to thank Gail Murphy for many useful comments.

I have been working in the Department of Computer Science, University of Tampere, over six years. Thanks to the supportive staff members of the department, working during those years has been so much fun. Special thanks to Teppo Kuusisto, Tuula Moisio, and Marja Liisa Nurmi for all their help.

Contents

Chapter 1

Introduction

The need for maintaining, reusing, and re-engineering existing software systems has increased dramatically over the past few years. Changed requirements or the need for software migration, for example, necessitate renovations for business-critical software systems. Reusing and modifying legacy systems are complex and expensive tasks because of the time-consuming process of program comprehension. Thus, the need for software engineering methods and tools that facilitate program understanding is compelling. A variety of *reverse engineering* tools provide means to support this task. Reverse engineering aims at analyzing the software and representing it in an abstract form so that it is easier to understand, e.g., for software maintenance, re-engineering, reuse, and documenting purposes.

To understand existing software systems, both static and dynamic information are useful. Static information describes the structure of the software as it is written in the source code, while dynamic information describes the run-time behavior. Both static and dynamic analysis result in information about the software artifacts and their relations. The dynamic analysis also produces sequential event trace information, information about concurrent behavior, code coverage, memory management, etc.

Program understanding can be supported by producing design models from the target software. This reverse engineering approach is also useful when constructing software from high-level design information, i.e., during *forward engineering*. The extracted static models can be used, for instance, to ensure that the architectural guidelines are followed and to get an overall picture of the current stage of the software. The dynamic models, in turn, can be used to support tasks such as debugging, finding dead code, and understanding the current behavior of the software.

The rise of new programming languages and paradigms 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++, Smalltalk, or Java. The adaption of the object-oriented programming paradigm has changed programming styles dramatically. Extracting information about the dynamic behavior of the software is especially important when examining object-oriented software. This is due to the dynamic nature of object-oriented programs: object creation, object deletion/garbage collection, and dynamic binding make it very difficult, and most times 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. One approach is to merge the extracted information into a single view and to support information filtering and hiding techniques and means to build abstractions in order to keep the view readable and understandable. However, when both static 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 formerly built static model. It is easy to add, e.g., information about code coverage to a static view but it is much more difficult to add information about concurrent or sequential behavior to that view. In addition, if a lot of information is attached to a single view it easily loses its readability.

Another approach to view the information extracted is to use different views and models for different purposes. For example, traditional message sequence charts (MSCs) [49] can be used to capture the interaction in a sample case, state diagrams to view the total behavior of the software, and static models to view the static software artifacts and their dependencies. Since static and dynamic models are distinguished in forward engineering, it is natural to do so also in reverse engineering. As in forward engineering, having separate views requires that there is a meaningful and consistent connection among these views. If such connections exist, the views can be used to comprehend each other, providing extended ways to support information exchange, slicing the views, and building abstractions. Furthermore, if the reverse engineering tool used is able to produce similar diagrams and models that have been used in the design phase of the software construction process, then an iterative software development approach that combines forward and reverse engineering techniques can be supported. Such software development is called *round-trip-engineering*.

SCED [56] is a prototype tool that has been built to support the dynamic modeling of object-oriented applications. It was originally designed to be used in analysis and design phases of the development process of object-oriented software. In this research, SCED is used to model the results of reverse engineering the run-time behavior of Java applications and applets. The main user interaction in SCED involves two independent editors: *a scenario diagram* editor and *a state diagram* editor. A scenario diagram in SCED is a variation of an MSC that semantically corresponds to a sequence diagram in Unified Modeling Language (UML) [95, 85]. A SCED state diagram notation can be characterized as a simplified UML statechart diagram notation. In SCED, state diagrams can be synthesized automatically from a set of scenario diagrams. The basic synthesis algorithm used was originally presented by Biermann and Krishnaswamy [7], and its adoption to state machine synthesis from scenarios is discussed by Koskimies and Mäkinen [54]. This algorithm with a few modifications has been implemented in SCED [56]. At any time during scenario editing the user can select one participating object and synthesize a state diagram automatically for it by using a single menu command. The state diagram can be synthesized from one scenario only or from a specified set of scenarios. Since the synthesis algorithm is incremental, scenarios can be synthesized to an existing state diagram. The synthesis algorithm is discussed in Chapter 5.

Several tools have been developed to visualize run-time behavior of object-oriented software systems [51, 59, 61, 99, 120]. Event traces are typically shown in a form of MSCs. In this research, the visualization of the run-time behavior has been taken one step further: not only SCED scenario diagrams but also the final specification of the dynamic behavior, i.e. the state diagram, is composed automatically as a result of the execution of a target system. This step is made possible by using the state diagram synthesis feature of SCED. Generated state diagrams allow the user to examine the dynamic behavior from a different angle compared to scenario diagrams. While scenario diagrams show the interaction among several objects, a state diagram shows the total behavior of a certain object or a method, disconnected from the rest of the system.

This dissertation shows that integration of dynamic and static information aids the performance of reverse engineering tasks. An experimental environment called Shimba has been built to support reverse engineering of Java software systems. The static information is extracted from Java byte code [118]. It can be viewed and analyzed with the Rigi reverse engineering tool [74]. The dynamic event trace information is generated automatically as a result of running the target system under a customized Java Development Kit (JDK) debugger. Information about the dynamic control flow of selected objects or methods can also be extracted. The event trace can then be viewed and analyzed with the SCED tool. To support model comprehension, the models built can be used to modify and improve each other by means of information exchange, model slicing, and building abstractions.

This dissertation is structured as follows. Reverse engineering approaches and tools are discussed in Chapter 2. Behavioral modeling with UML is briefly discussed in Chapter 3. Chapter 4 gives an overview of the SCED tool and describes its diagrams used for dynamic modeling, comparing them to the ones used in UML. In Chapter 5, the state diagram algorithms presented by Koskimies and Mäkinen are introduced with few modifications caused by the extended scenario notation of SCED. The synthesized state diagram can be simplified by adding UML statechart diagram concepts into it. The simplifying methods are introduced in Chapter 6. The Rigi tool and its reverse engineering methodology are briefly discussed in Chapter 7. The reverse engineering approach and features of Shimba are described in Chapter 8. To validate the usability of the approach, explained in Chapter 8, a target Java software system is examined. The results and examples of this case study are presented in Chapter 9. This research is related to other work in Chapter 8.10. Finally, Chapter 10 discusses the research, highlights the contributions, and addresses some future plans.

Chapter 2

Reverse engineering

Chikofsky and Cross [18] define reverse engineering as a process of analyzing a subject system with two goals in mind:

(1) to identify the system's components and their interrelationships and

(2) to create representations of the system in another form or at a higher level of abstraction.

Reverse engineering aims to support program comprehension. Reverse engineering approaches can thus facilitate, for example, maintenance, reuse, documentation, re-engineering, and forward engineering of the target software. Program comprehension can be supported by producing design models from existing software. In this dissertation, modeling the static structure of the target software is called *static reverse engineering*, and modeling its dynamic behavior is called *dynamic reverse engineering*.

Reverse engineering is difficult for various reasons. First, the target software can be, and often is, poorly documented. In addition, the documentation is seldom up to date. Second, persons who designed and implemented the software cannot always be reached for consultation. Such difficulties often mean that the only reliable source of information is the source code. Third, there is a gap between the top-down process often used in a forward engineering process and the bottom-up analysis of the source code typically used in static reverse engineering. Deriving similar models from source code as were used in the design phase of the forward engineering process is difficult and in many cases impossible. For example, a Java software system can be designed using UML. Code generators can even be used to construct skeletons of classes automatically. However, there is no one-to-one correspondence between UML modeling concepts and Java software artifacts. For instance, aggregation and composition do not have direct counterparts in Java and, vice versa, method bodies cannot be expressed in UML. Fourth, the functionality and purpose of some structures used in the source code might be difficult to understand. Such structures can be technical and/or language dependent solutions to implementation problems. Fifth, the source code includes both domain dependent and domain independent code. The former is especially problematic, forcing the engineer to become familiar with the domain as well. Sixth, combining results of dynamic reverse engineering and static reverse engineering is difficult, especially for examining object-oriented software systems. Object-oriented programs are inherently dynamic: object creation, object deletion/garbage collection, and dynamic binding cause behavior that is difficult, and often impossible, to understand by just examining the source code. Thus, dynamic reverse engineering is especially important for understanding object-oriented software systems. For the reasons above, automating the tedious task of reverse engineering is especially difficult.

Chikofsky and Cross [18] further characterize *design recovery* as a subset of reverse engineering in which domain knowledge, external information, and deduction or fuzzy reasoning are added to the observations of the subject system. The objective of design recovery is to identify meaningful higher-level abstractions beyond those obtained directly by examining the system itself.

2.1 Extracting and viewing information

All reverse engineering environments need tools for extracting the information to be analyzed. Static information includes 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, calls between methods, and so on. The static reverse engineering process may also include syntax and type checking, and control and data flow analysis [2]. Dynamic information contains software artifacts as well. In addition, it contains sequential event trace information, information about concurrent behavior, memory management, code coverage, etc. Static information can be extracted, e.g., by using parsers based on grammars. For extracting dynamic information, debuggers, profilers, or event recorders can be used. In addition, source code instrumentation is an often used approach. Furthermore, when analyzing languages like Java or Smalltalk, the instructions of the virtual machine (VM) can be instrumented instead.

The extracted information is not useful unless it can be shown in a readable and descriptive way. Supporting program comprehension by building (graphical) design models from existing software is supported in many reverse engineering and design recovery tools and environments. There are basically three kinds of views that can be used to illustrate the extracted information: static views, dynamic views, and merged views. Static views contain only static information, dynamic views contain only dynamic information, and merged views are used to show both static and dynamic information in a single view. Figure 2.1 shows different choices of building views to the target software.

2.1.1 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 ensured when merging static and dynamic information. For example, because of polymorphism, a static analysis is not enough to conclude the exact method calls; a method call written in the source code represents a set of possible operations, rather than a certain single operation that is invoked at run-time. Dynamic analysis is needed to determine the actual method calls.

Building abstractions for merged views can be difficult because static and dynamic abstractions usually differ considerably. While static abstractions are subsystems, dynamic abstractions are typically use cases or behavioral patterns (i.e., repeated similar behavior). The user therefore has

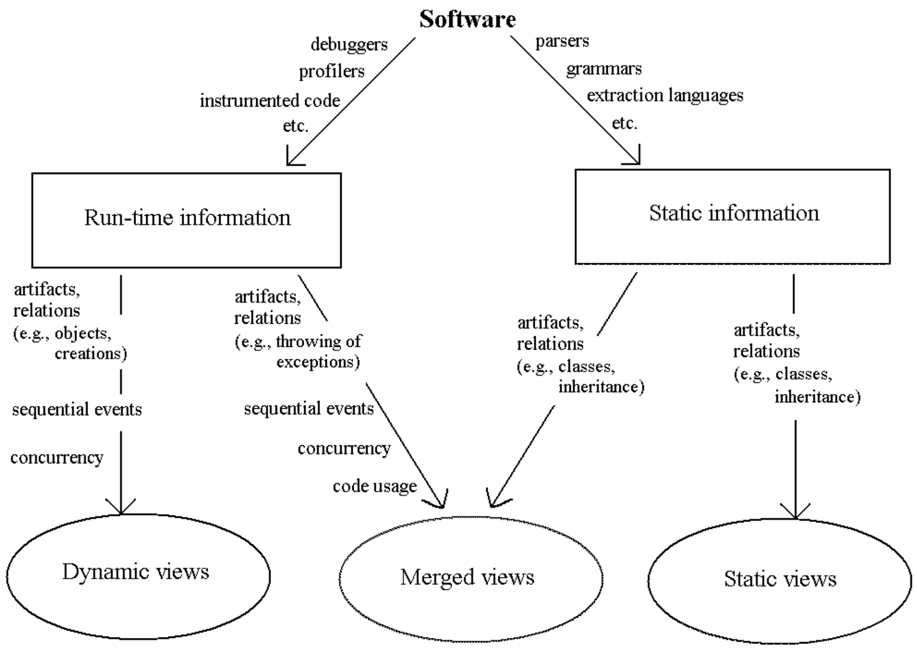


Figure 2.1: Different choices of constructing views to the target software

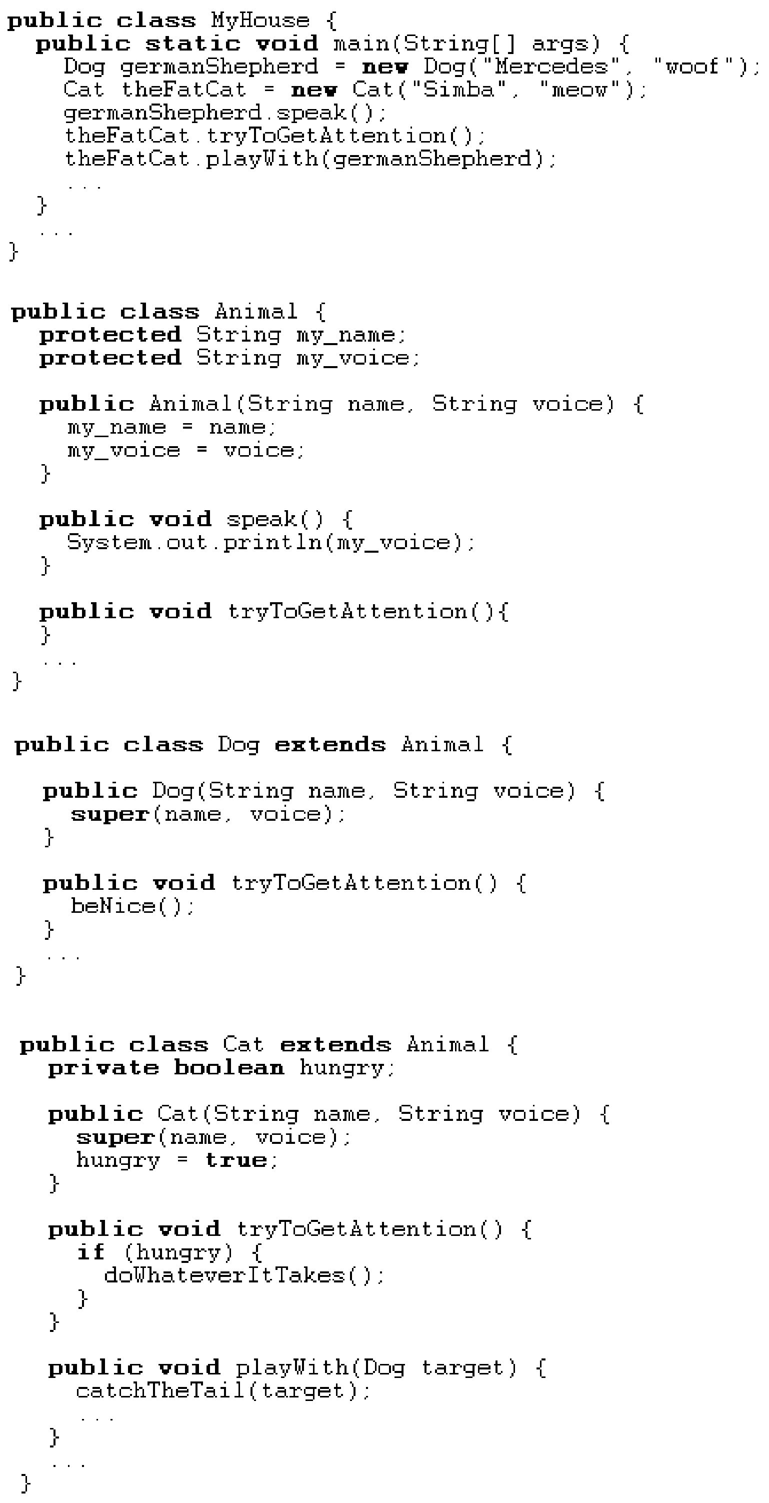
to choose at an early stage whether to build the abstractions from a static or dynamic point of view. For example, consider a banking system that consists of banks, consortiums of banks, and ATMs. An ATM can be used, e.g., for withdrawing cash or for paying bills. From a static point of view, an ATM, a consortium, and a bank themselves represent subsystems. From a dynamic point of view, in turn, “withdrawing money using an ATM” and “paying a bill using an ATM” are two different use cases, both representing communication among ATM, consortium, and bank subsystems.

Forming merged views themselves might be complicated. For example, it is easy to add code coverage information that shows the actual run-time usage of the software artifacts to a static view but it is much more difficult to add information about concurrent or sequential behavior to it. In UML, *collaboration diagrams* can be used to view both dynamic event trace information and static aspects of the software. 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. To focus on desired aspects of the software, uninteresting information can be filtered out or hidden. On the other hand, if such techniques provides the only means to focus on the chosen aspect of the software, e.g., sequential event trace information, then merging that information into the view is questionable. Unless the merge serves another purpose, choosing a more suitable and descriptive view would probably promote the reverse engineering task better.

2.1.2 A set of different views

Figure 2.2 shows the source code of an example Java program. When reverse engineering the example program, the static information could be shown as a *class diagram* as depicted in Figure 2.3. The class diagram shows the static model elements of the subject program, as well as their contents and relationships. The dynamic behavior could be visualized as a *scenario diagram*, which describes the object interactions. Time (or execution) in the scenario diagram flows from top to bottom. Figure 2.4 shows a SCED scenario diagram that could characterize the dynamic behavior of the example Java program.
In forward engineering different diagrams are used to model the static structure and dynamic behavior of the software system. For instance, in UML there are static diagrams, dynamic diagrams, and diagrams that model both the 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 also be the case 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 the previous section occur. The number and type of diagrams to be used depend on the purpose and needs in the same way as in forward engineering.

Separating 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 better possibilities to support 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 (i.e., only a desired part of the static model is shown). The resulting slice shows the structure of a particular



```
public class MyHouse {
  public static void main(String[] args) {
    Dog germanShepherd = new Dog("Mercedes", "woof");
    Cat theFatCat = new Cat("Simba", "meow");
    germanShepherd.speak();
    theFatCat.tryToGetAttention();
    theFatCat.playWith(germanShepherd);
    ...
  }
  ...
}

public class Animal {
  protected String my_name;
  protected String my_voice;

  public Animal(String name, String voice) {
    my_name = name;
    my_voice = voice;
  }

  public void speak() {
    System.out.println(my_voice);
  }

  public void tryToGetAttention(){
  }
  ...
}

public class Dog extends Animal {

  public Dog(String name, String voice) {
    super(name, voice);
  }

  public void tryToGetAttention() {
    beNice();
  }
  ...
}

public class Cat extends Animal {
  private boolean hungry;

  public Cat(String name, String voice) {
    super(name, voice);
    hungry = true;
  }

  public void tryToGetAttention() {
    if (hungry) {
      doWhateverItTakes();
    }
  }

  public void playWith(Dog target) {
    catchTheTail(target);
    ...
  }
  ...
}
```

Figure 2.2: The source code of an example Java program

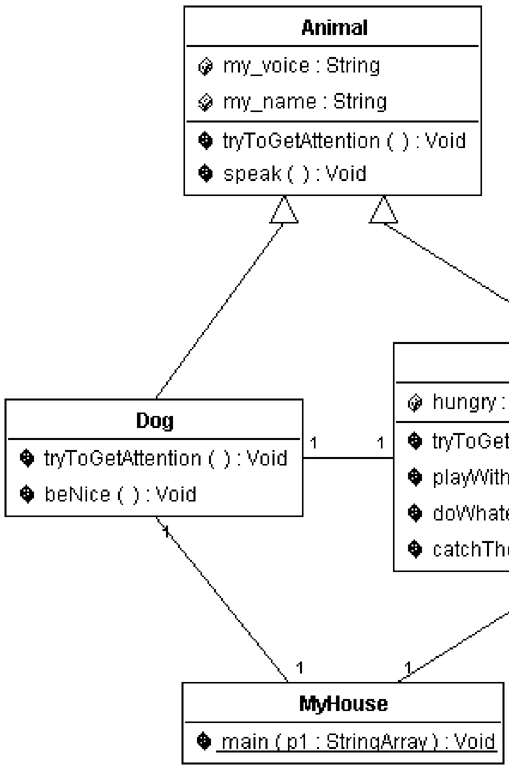

Animal
my_voice : String
my_name : String
tryToGetAttention ( ) : Void
speak ( ) : Void
Dog
tryToGetAttention ( ) : Void
beNice ( ) : Void
Cat
hungry : Boolean
tryToGetAttention ( ) : Void
playWith (target : Dog) : Void
doWhateverItTakes ( ) : Void
catchTheTail (target : Dog) : Void
MyHouse
main (p1 : StringArray) : Void
1
1
1
1
1
1

Figure 2.3: The static structure of the program in Figure 2.2 is shown as a class diagram.

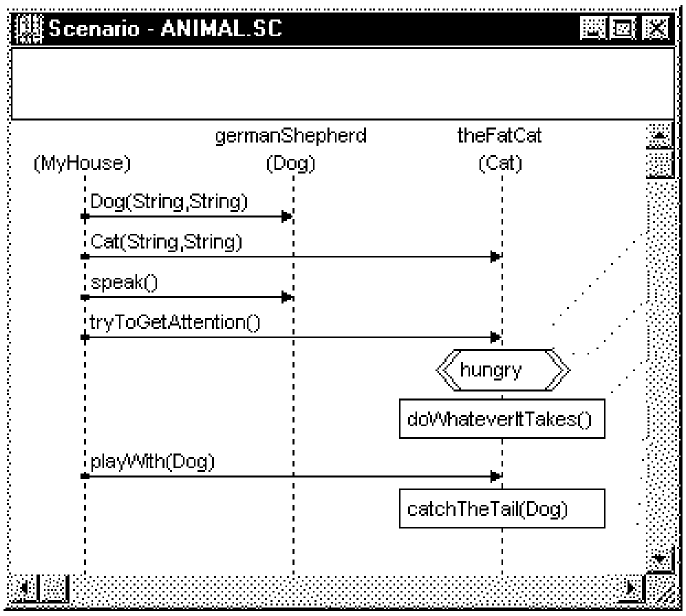


Figure 2.4: The program in Figure 2.2 has to be executed to capture its dynamic behavior. A scenario diagram can be used to visualize the execution.

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.

Using a set of different views makes it possible to build abstractions for dynamic views according to different principles than 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 criterion 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 among high level static components instead of showing the interaction between classes or even objects.

2.2 Reverse engineering approaches and tools

A wide range of reverse engineering and design recovery tools have been developed for both industrial use and academic research. Most of them provide better support for static reverse engineering than for dynamic reverse engineering. Some of the tools focus on understanding the software by building high-level models of the structure and/or the behavior of the software, some tools can be used to analyze the software based on software metrics and other measurements, and some tools support re-engineering and round-trip-engineering by providing facilities for both forward and reverse engineering of the software. There are also tool sets that support all these approaches.

In what follows, we briefly describe different reverse engineering and design recovery approaches and give examples of tools and tool sets that support these approaches.

2.2.1 Understanding the software through high-level models

Tools that extract static and dynamic information from the target software typically produce a lot of detailed information. Hence, good views for showing that information is not usually enough, but abstractions need to be built for making the views clearer and more understandable. In static reverse engineering, abstract high-level components to be found and constructed might represent subsystems or other logically connected software artifacts. In dynamic reverse engineering, abstractions are typically behavioral patterns, use cases, or views that show interaction among high-level static components.

Constructing abstract and descriptive high-level views of the target software is the most challenging phase in the reverse engineering process described in Figure 2.1. Gathering information and building the initial views are not straightforward either: an empirical study by Murphy *et al.* compares nine static call graph extractors and shows considerable differences among the results obtained from three C software systems [72]. The main reason for this was that the requirements for tools computing call graphs are typically more relaxed than those for compilers. In general, the information can be extracted and initial views of the software can be constructed automatically. However, manual processing is needed in most cases for building high-level views from the detailed low-level views. In static reverse engineering, language structures and metrics can be used to partly automate the process. There are slightly more efficient ways to automate the construction of abstract dynamic views. For example, pattern matching algorithms can be used to automatically search for behavioral patterns. Furthermore, abstractions are typically constructed for the static views before constructing them for the dynamic views. The static hierarchies can then be used for clustering the dynamic information automatically (cf. Sections 8.9 and 9.4.3).

Most of the static reverse engineering tools and environments use graphical representations to view the extracted information. Some of the tools allow manipulations of the view/views and give support for building high-level models of the target software to facilitate program comprehension. Next we give examples of such tools. An introduction of six static reverse engineering or design recovery tools is followed by a description seven tools that emphasize dynamic reverse engineering. The tools are selected to give examples of unique categories of reverse engineering and design recovery approaches.

The Rigi reverse engineering environment [74], for example, uses a directed graph to view the software artifacts and their relations and supports the extraction of abstractions and design information out of existing software systems [73]. To build more abstract views to the software, the user can form hierarchical structures for the graph by using subsystem composition facilities supported by the graph editor. Such structures are shown as nested views. Rigi is discussed in Chapter 7 in more detail.

Since Rigi is easy to customize, tailor, and extend, it has been integrated with several other tools and environment, for example, the Portable Bookshelf (PBS) [34] and the Dali [52] tool sets. The PBS is intended to be developed, managed, and used by three types of people: a builder, a librarian, and a patron. A builder creates the bookshelf architecture. She designs a general program-understanding schema and integrates usable tools to support a librarian in her work. A librarian populates the bookshelf repository with information about the target software system. Finally, a patron is an end-user of the bookshelf content who needs detailed information to re-engineer the legacy code [34].

Dali is a workbench for architectural extraction, manipulation, and conformance testing [52]. It integrates several analysis tools and saves the extracted information in a repository. Dali uses a merged view approach, modeling all extracted information as a customized Rigi graph. In addition to static information, the constructed Rigi graph contains information about the behavior of the target software system, extracted using profilers and test coverage tools. The user can organize and manipulate the view and hence produce other, refined views on a desired level of abstraction.

Imagix4D from Imagix Corporation [46] supports reverse engineering and documenting C and C++ software systems. The source code of the target software can be analyzed and browsed at any level of abstraction using different views. Imagix4D uses 3D views to help the user to focus and analyze particular aspects of the software.

DESIRE [8] is a model-based design recovery system that can be used for concept recognition and program understanding. It provides intelligent assistant facilities to search for instances of user-defined concepts, to identify concepts that correspond to some domain model concept, and to propose a concept assignment for a given interest set. DESIRE is also able to produce call graphs, reference points of global variables, symbols defined in a given scope, filterings and clusterings of components and dependencies, etc.

ManSART is a software architecture recovery system that uses an abstract syntax tree (AST) of the program as a source of information [14]. The AST is produced using Refine-based workbenches by Reasoning Systems [86]. With ManSART the user is able to interpret and integrate the results of localized, perhaps language-specific, source code analysis in the context of large size systems written in multiple languages [14].

Dynamic reverse engineering tools often use variations of a basic MSC or directed graphs to visualize the run-time behavior of the target software system. For example, a directed graphs can be used to visualize the run-time object interactions by representing objects as nodes and visualizing method calls or variable accesses as arcs between the nodes. Both of these graphical representations are simple and self-explanatory and thus suitable to be used for program understanding purposes. However, without notational extensions, they do not scale up. A large amount of run-time information is typically generated, even as a result of a relatively brief usage of the system. Thus, managing and abstracting the extracted information is necessary. This is usually the most challenging problem in dynamic reverse engineering. Behavioral patterns are often used to build abstract views of the dynamic event trace information. High-level views can also be constructed by taking advantage of abstractions built for the static view. Both of these approaches are used in this research.

Ovation uses *execution pattern views* to visualize and explore a program's execution at different levels of abstraction [26, 27]. It offers several means to manipulate the view, e.g., for raising the level of abstraction and to manage the event explosion problem.

Sefika *et al.* introduce an architectural-oriented visualization approach that can be used to view the behavior of a target system in different levels of granularity [99]. 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 include subsystem, framework, pattern, class, object, and method levels.

Walker *et al.* use high-level models for visualizing program execution information [120]. In the main view, called *a cel*, high-level software components are represented as boxes. The mapping between low-level software artifacts and high-level components they belong to is done manually using a declarative mapping language. The visualization technique by Walker *et al.* also focuses on showing summary information (e.g., current call stacks and summaries of calls).

Scene tool produces and visualizes event traces as scenario diagrams [59]. It allows the user to browse the scenarios and other associated documents. For compressing the large amount of extracted event trace information Scene shows the operation calls (messages) in a closed form as default: the internal events of a call are not shown unless ’opened’ by clicking the call arc. In this way the user can proceed to the interesting level, in a top-down fashion.

ISVis is visualization tool that supports the browsing and analysis of execution scenarios [51]. In ISVis, the event trace can be analyzed using a *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.

Program Explorer combines static information with run-time information to produce views that summarize relevant computations of the target system [60, 61]. It uses directed graphs to illustrate class relationships and object interactions. The order of the interactions is viewed as *interaction charts*. To reduce the amount of run-time information generated the user can choose when to start and stop recording events during the execution. Merging, pruning, and slicing techniques are used for removing unwanted information from the views.

Richner *et al* present a query-based approach to recover high-level views of object-oriented applications [87]. Static and dynamic aspects of the target software are modeled in terms of logic facts. Depending on the queries made, the views may 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.

A *design pattern* systematically names, explains, and evaluates and important and recurring design in object-oriented design. Each pattern describes a frequently occurring problem and describes the core of the solution to it. Gamma, Helm, Johnson, and Vlissides have catalogued and described several popular creational, structural, and behavioral design patterns [36]. Tools that support the identification of the design patterns help engineers to learn and understand object-oriented software systems. Bansiya introduces the DP++ tool that automates design-pattern detection, identification, and classification in C++ programs [5]. The DP++ tool identifies several structural and behavioral patterns.

2.2.2 Software metrics

Software metrics have traditionally been used in forward engineering to improve the quality of the software. For example, software metrics can be used to measure the complexity of the software design and to predict properties of the final product. They can also be used to predict the amount of testing necessary or the total development costs [25].

Software metrics can play a significant role also in the reverse engineering process. Complexity metrics can be applied to support the identification of complex parts of the software. Such parts typically need restructuring to improve the reusability and the reliability of the software. One of the most commonly used complexity measure is cyclomatic complexity [70]. It has been widely used in various reverse engineering environments and applied as the basis for other metrics.

Design flaws can also be identified by applying appropriate metrics. Metrics for object interactions can reveal tightly coupled and/or loosely cohesive parts of the software [16, 17, 39]. Tightly coupled parts are inflexible for modifications and reuse. Loosely cohesive parts might also need restructuring. For example, low cohesion inside a class in an object-oriented software system might hint that the class contains unfitting or unused methods or variables.

Metrics that examine the inheritance hierarchy of object-oriented software systems are used to predict reusability and complexity of the software. For example, deep inheritance trees constitute greater design complexity since more methods and classes are involved in dynamic binding. On the other hand, they provide more choices for potential reuse.

Li and Henry have used software metrics that focus on inheritance hierarchy, complexity, coupling, and cohesion to measure maintainability in two independent empirical studies [64, 65]. Some of the metrics can be applied to software written in any language, while others are dependent on the programming paradigm or the language. For example, *object-oriented metrics* [66, 43] are used to evaluate object-oriented software systems.

Software metrics are used in many reverse engineering environments to help the user to analyze constructed views of the target software. In Rigi, a “low coupling and high cohesion” principle is used for subsystem structure identification when reverse engineering C programs [73]. McCabe Reengineer from McCabe & Associates Inc. [71] provides views of the system architecture and views of the interaction among modules, based on the analysis of the source code. Complexity and structuredness of software modules is measured using metrics. The results are shown using a specific coloring on the views.

CodeCrawler is a platform built to support program understanding by combining metrics and program visualization [28]. CodeCrawler provides views that show selected structural aspects of the software as a simple two-dimensional graph. A node in a graph represents a software artifact in C++ code (e.g., a class). CodeGrawler is able to visualize up to five metric values simultaneously on a single node: the size of a node can render two measurements (the width and the height), the position of the node can also render two measurements (X and Y coordinates), and the color of the node that may vary between white and black can be used to visualize one measurement.

Hindsight reverse engineering tool from IntegriSoft Inc. is able to produce different kinds of reports, charts, and diagrams that help program understanding [48]. Hindsight uses software metrics to analyze the complexity of the target software. It also supports dynamic testing of the software. The dynamic information is generated using a source code instrumentation technique.

Logiscope from CS Verilog supports both static and dynamic analysis of a target software system [23]. It is able to produce static call and control graphs of the target software. Quantitative information based on software metrics and graphs can be generated to help the user to diagnose defects. For dynamic analysis of a target software system Logiscope provides the TestChecker tool to measure structural test coverage and to detail the uncovered source code paths. TestChecker uses source code instrumentation approach to generate the dynamic information.

2.2.3 Supporting re-engineering and round-trip-engineering

Chikofsky and Cross characterize re-engineering as an examination of a subject system to reconstitute it in a new form and the subsequent implementation of the new form [18]. Reverse engineering approaches are typically used for understanding the subject system in a re-engineering process. However, reverse engineering techniques can and should be applied for forward engineering as well. That would support a change from a conventional “water fall” style of forward engineering to a more incremental and evolutionary style of software construction. In other words, round-trip-engineering would be supported. To support re-engineering and round-trip-engineering a reverse engineering tool should be able to produce standard object-oriented analysis and design (OOAD) models from the target software. This would give the user an obvious benefit: since such models are (probably) familiar to the user from designing the software, using them for reverse engineering would unburden her from learning yet another model or diagram notation.

Various tools supporting forward engineering of object-oriented software are also able to extract class diagrams for existing software systems, for example, Rational Rose from Rational Software Corporation [82, 83, 84], Paradigm Plus from Computer Associates International [22], OEW from Innovative Software GmbH [47], Graphical Designer from Advanced Software Technologies Inc. [1], Domain Objects from Domain Objects Inc. [29], COOL:Jex from Sterling Software Inc. [105], etc. To give full support for round-trip engineering extraction of class diagrams is not enough. It is far more difficult to construct dynamic models like UML statechart diagrams and use case diagrams from the recorded run-time behavior than to generate class diagrams from the source code. As discussed in Section 8.10.1, dynamic reverse engineering tools typically use directed graphs or variations of an MSC to visualize the run-time behavior. In this research, not only SCED scenario diagrams but also state diagrams are used for modeling the run-time behavior (cf. Chapter 8). However, the ultimate goal of constructing state diagrams was supporting program understanding, rather than supporting round-trip-engineering. Hence, state diagrams are used for understanding the behavior of a target Java software system, not for specification of a software system to be implemented.

Versatile tools and environments that support both forward and reverse engineering are available. StP from E2S is a modeling-based software development environment that also supports reverse engineering, testing, and requirements engineering [33]. StP provides different tool sets for developing and maintaining software written in different languages. For example, StP/UML, StP/OMT, and StP/Booch integrated with third-party programming environments can be used for incremental code generation and reverse engineering of object-oriented software systems.

The Viasoft Existing Systems Workbench (ESW) from Viasoft Inc. is an integrated software tool set that supports software maintenance in various ways [119]. The tool set includes, for example, a reengineering tool Renaissance, a static analysis and a documentation generator SmartDoc, application and program understanding and visualization tools Alliance and Insight, a software testing and debugging tool SmartTest, a code generation and converting tool AutoChange, and a metrics tool Recap.

Tool sets Ensemble and ObjectTeam from Sterling Software Inc. support application development of C and object-oriented programs, respectively [105]. Ensemble provides graphical views for analyzing the design of the target software. Complexity metrics can be applied to the software to help the designer to make a re-design or re-use decision. Ensemble also supports testing and documentation generation.

2.2.4 Other tools facilitating reverse engineering

Reverse engineering a target software system can be supported in several ways. As discussed above, design models can be constructed to characterize the structure and the behavior of the software visually, while metrics can be used to point out its interesting aspects or design flaws. Tools that support browsing the documentation or source code also support program comprehension. Hypersoft tool supports automated detection of software structures that are critical for understanding and re-engineering C software systems [77]. It also enables the navigation of such structures through automatically generated hypertext documents. Furthermore, the Hypersoft tool supports the examination of the side effects of software renovations, detecting errors, and controlling the testing of the re-engineered software. Tools that support other reverse engineering tools form yet another interesting group of tools. Software Refinery from Reasoning Systems, for example, is a set of tools that can be used to generate reverse other engineering tools [86]. It contains tools for generating source code parsing and conversion tools. Software Refinery supports C, Ada, and Cobol.

2.2.5 Summary

Both static and dynamic reverse engineering are needed to understand an object-oriented software system fully. Compared to procedural languages, the importance of dynamic reverse engineering needs to be emphasized when studying object-oriented software systems. This is due to the dynamic nature of object-oriented programs. The extracted information needs to be shown in a readable and descriptive way. Static and dynamic information can be presented in separated views or merged in a single view. Both approaches have advantages and disadvantages. In this research, the multiple view approach is promoted.

A wide range of reverse engineering and design recovery tools can be categorized in various ways. We identify the following three groups: tools that support program understanding through high-level models, tools that use software metrics for studying software properties, and tools that support re-engineering and round-trip engineering. The Shimba environment presented in this dissertation belongs to the first group.

Chapter 3

Modeling with UML

The Unified Modeling Language (UML) has been accepted as an industrial standard for specifying, visualizing, understanding, and documenting object-oriented software systems [95, 85]. It provides several diagram types that can be used to view and model the software system from different perspectives and/or at different levels of abstraction. UML supports all lifecycle stages of the forward engineering process from requirements specification to implementation and testing. The same diagram types used in forward engineering have been used for reverse engineering purposes as well [1, 29, 83, 84, 105].

First object-oriented analysis and design (OOAD) methods were published in the late 80's and early 90's. In addition to the three independent core methods of UML, namely Booch '91 [9], object-oriented modeling and design (OMT-1) [94], and object-oriented software engineering (OOSE) [50], methods were published, e.g., by Coad and Yourdon [19], Shlaer and Mellor [98], and Wirfs-Brock *et al.* [122]. The development of UML began in 1994. The first draft called Unified Method 0.8 was released in 1995. It merged second editions of Booch '91 and OMT-1, namely Booch '93 [10] and OMT-2 [90, 91, 92, 93]. When OOSE was merged into the Unified Method in 1996, the name was changed to UML. The first official version, UML 1.0, was published in 1997, followed by versions 1.1 and 1.3. The evolution of UML is depicted in Figure 3.1. Another attempt to join different OOAD methodologies was Fusion [20], which included concepts of OMT, Booch '91, and CRC [122].

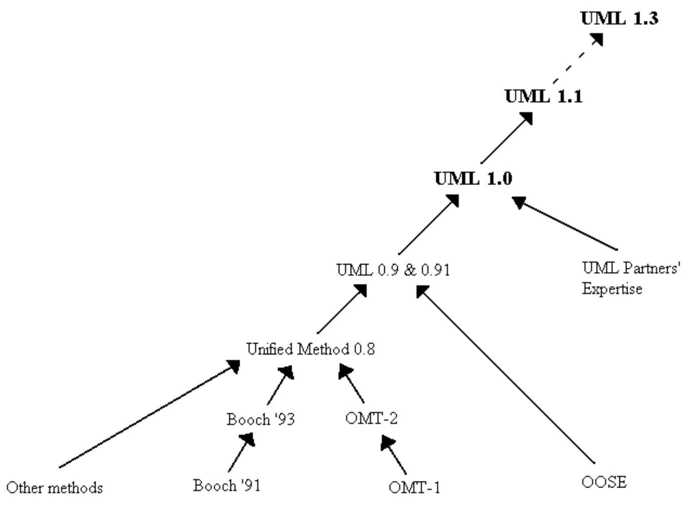


Figure 3.1: Evolution of UML[85]

UML provides diagrams that capture information about the static structure of the software, and diagrams that model the dynamic behavior of the software. Some of the diagrams (e.g., *a collaboration diagram*) combine both dynamic and static aspects of the software. UML also contains organizational constructs for managing and arranging other models. Furthermore, UML provides concepts and general model elements that can be used to make some common extensions without changing the underlying modeling language and concepts and general model elements that can be used to extend different models. Table 3.1 shows the diagram types of UML.

Next we discuss selected UML diagrams, starting from class diagrams. Since the focus of this research is on dynamic modeling, the rest of this chapter discusses behavioral modeling using UML, the emphasis being on sequence diagrams and statechart diagrams. Collaboration diagrams and activity diagrams are also briefly characterized.

Diagram types	Diagrams
Static structure diagrams	class diagram object diagram
Use case diagrams	use case diagram
Behavioral diagrams	statechart diagram activity diagram sequence diagram collaboration diagram
Implementation diagrams	component diagram deployment diagram

Table 3.1: Different diagram types of UML[95]

3.1 Class diagrams

A *class diagram* is a graphical presentation of the static view that shows a collection of declarative (static) model elements, such as classes, interfaces, types, as well as their contents and relationships [85, 95]. In what follows, we discuss the main parts of the class diagram notation.

A *class* is the descriptor for a set of objects with similar structure, behavior, and relationships. A class is drawn as a rectangle with three compartments separated by horizontal lines. The top compartment holds the class name. The middle and bottom compartments are reserved for a list of attributes and a list of operations, respectively. An *interface* is a named set of operations that characterize the behavior of an element [95]. Interfaces are shown as rectangles with two compartments. The top compartment shows the name of the interface and includes a stereotype “<<interface>>”. The bottom compartment contains the list of operations. Besides classes and interfaces, a class diagram may also contain, for instance, packages and types.

Various kinds of relationships may exist among model elements of a class diagram. An *association* between two or more classes indicate that there are connections among instances of the classes. The connections can be, for example, method calls or links between the objects. An association between two classes is shown as a solid line connecting the rectangles of the classes. Additional information can be attached to an association. For example, an association may be directed and it may show the multiplicity and rolenames of instances involved in the connection at each end of the associations. *Generalization* relationships can be used to show inheritance between classes. Generalization is depicted as a solid line from the subclass to the superclass, with a large hollow triangle at the end of the superclass. *Composition* is a form of aggregation with strong ownership and coincident lifetime [95]. Composition may be shown as a solid filled diamond at the end of the owner class.

Figure 3.2 shows a simple class diagram describing an elevator system. The system consists of five classes. Class *Janitor* inherits class *Person*, indicating that a janitor is a person. In addition to four operations inherited from class *Person*, operation *maintain()* can be called for each instance of class *Janitor*. Classes *Elevator* and *House* have a composition relationship. It indicates that an elevator is a part of a house. The multiplicities of the composition defines the situation more specifically: in a house there can be up to four elevators and a particular elevator can be in one house only. Other relationships are normal associations. The class diagram has been drawn using the FUJABA tool [88].

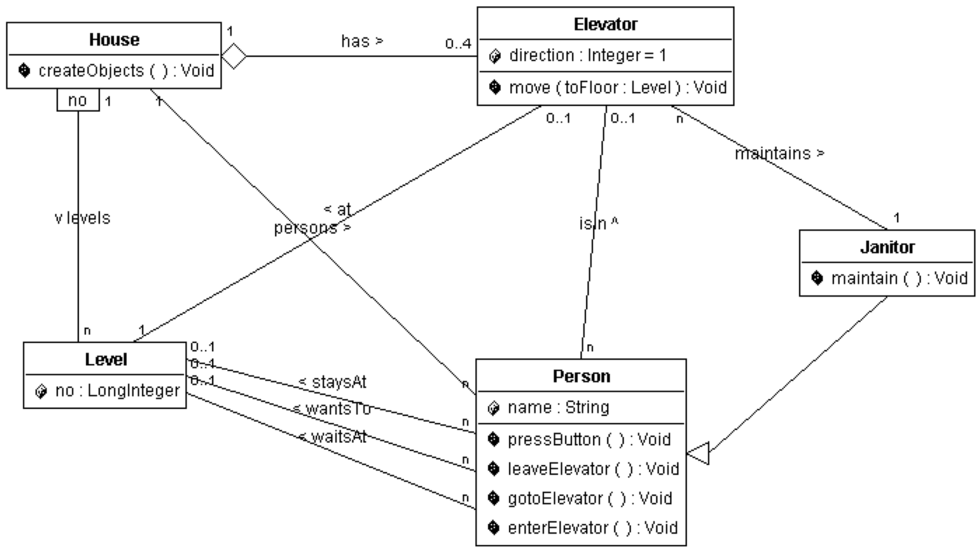


Figure 3.2: A class diagram describing an elevator system. The class diagram has been constructed using the FUJABA tool[88].

3.2 Sequence diagrams

A *sequence diagram* describes the object interaction arranged in time sequence. Participating objects are shown by their *lifelines* as vertical lines. A lifeline shows the existence of an object over a period of time. For any period during which the object is active, the lifeline is broadened to a double solid line. *Messages* exchanged by objects are drawn as arrows between lifelines. A message is a conveyance of information from one object to another, with the expectation that an activity will ensue [95]. It may be a signal or a call of an operation. The receipt of a message instance is normally considered an instance of an event, which is a specification of a noteworthy occurrence that has a location in time and space [95]. Sequence diagrams occur in slightly different formats when intended for different purposes [85]. Two examples of sequence diagrams are given in Figures 3.3 and 3.4. Figure 3.3 shows a simple sequence diagram with three concurrent objects. Comments are written on the left of the diagram as plain text. Timing constraints are closed inside braces. The sequence diagram in Figure 3.4 contains the following additional UML sequence diagram concepts: an object creation (e.g., *op()* creates an object *ob1*), conditional branching (events *[x $>$ 0] foo(x)* and *[x $<$ 0] bar(x)*), conditional branches in the communication (branching dotted line of *ob4:C4*), a recursion (the object *obj1* calls its own *more()* method), and an object deletion (crosses at the end of lifelines of *ob1:C1* and *ob2:C2*). Branching shown as multiple arrows leaving a single point may represent conditionality or concurrency, depending on whether the guard conditions are mutually exclusive or not [85]. The branching in Figure 3.4 hence represents conditionality.

3.3 Collaboration diagrams

A *collaboration diagram* shows an interaction organized around objects (needed in the interaction) and their links to each other. A collaboration diagram is very close to a sequence diagram. They both show interactions, but they emphasize different aspects. A sequence diagram shows the interaction over time but does not show other relationships among objects than the messages belonging to the interaction. A collaboration diagram, in turn, does not show time as a separate dimension. The order of messages can be expressed by numbering. The relationships among the objects are

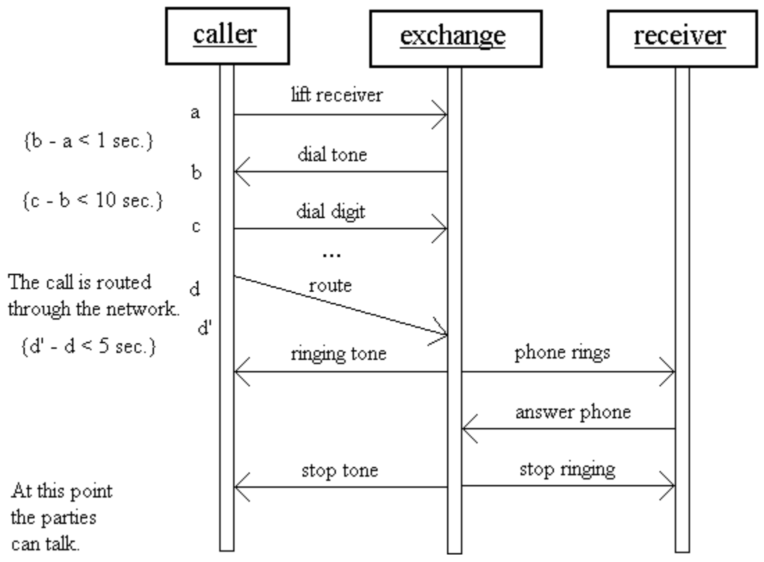


Figure 3.3: A simple sequence diagram with concurrent objects [85] (Notation Guide)

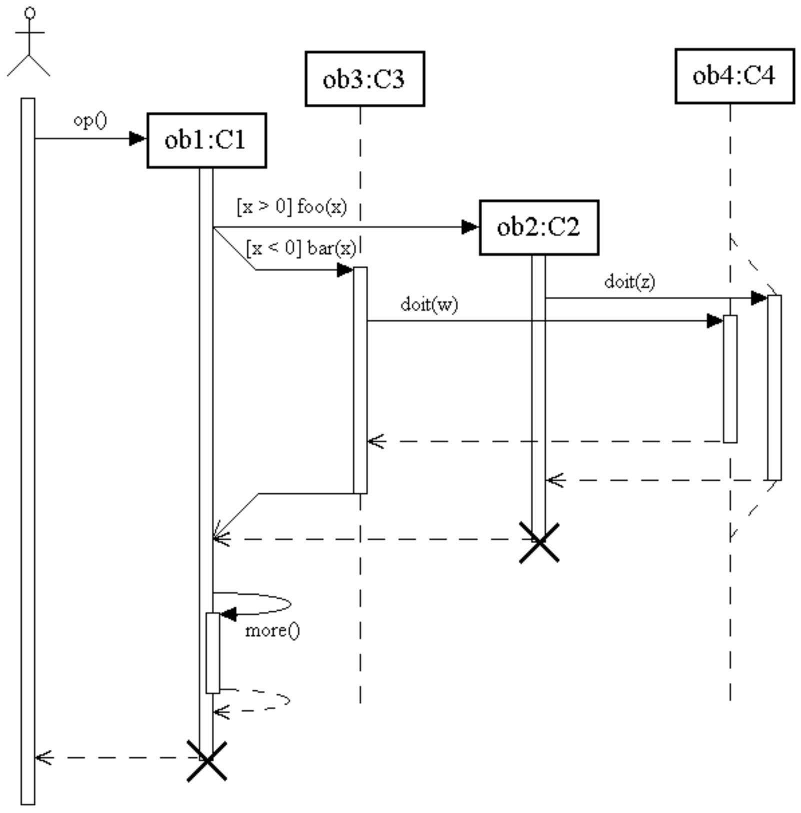


Figure 3.4: A sequence diagram with focus of control, conditional, recursion, creation, and destruction [85] (Notation Guide)

explicitly shown in a collaboration diagram. Hence, a collaboration diagram also includes a static aspect. While sequence diagrams show the explicit sequence of stimuli and are hence better for realtime specification and complex scenarios, collaboration diagrams show the full context of an interaction, including objects and relations relevant to a particular interaction [85]. Figure 3.5 shows an example of a collaboration diagram.

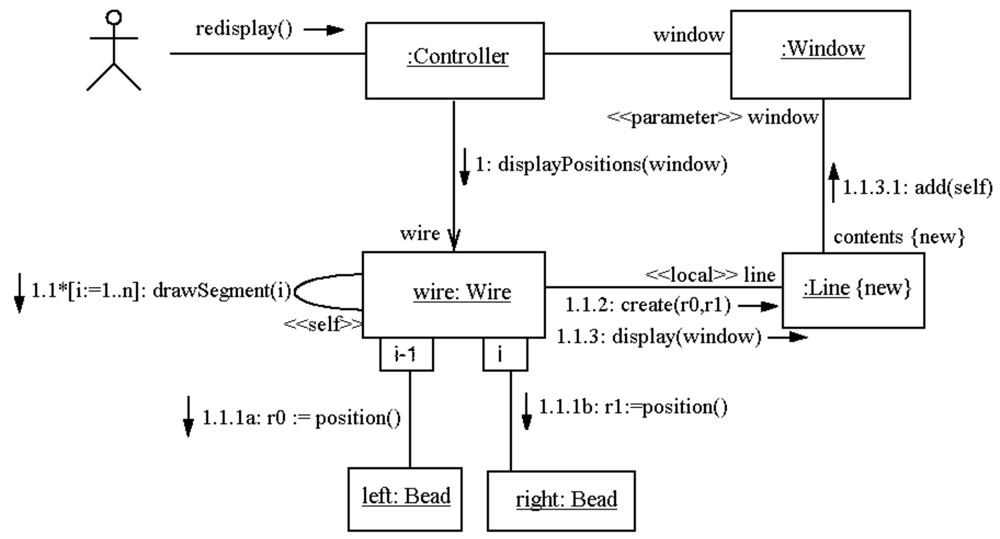


Figure 3.5: A collaboration diagram with message flows[85](Notation Guide)

3.4 Statechart diagrams

A *state machine* of an object is a directed graph that consists of states and transitions, describing the response of the object to external stimuli. A *statechart diagram* is a graph that represents a state machine. The semantics and notation used in UML follow Harel's statecharts [40]. Statecharts are a widely used notation for structuring state machines and avoiding the combinatorial explosion that plagues them. Harel's statecharts play a significant role in the design process of a larger development methodology that has been implemented as a commercial product called STATEMATE [41, 42] from I-logic Inc. STATEMATE is a set of tools used for modeling reactive systems. STATEMATE is most beneficial in requirements analysis, specification, and high-level design [42]. Rhapsody is another tool from I-Logic, in which Harel's statecharts are used. The Rhapsody tool can be used for analyzing, modeling, designing, implementing, and verifying the behavior of embedded systems software. Prior to UML, statecharts have been adopted by other OOAD methodologies as well, including OMT. The use of statecharts in object-oriented design is discussed by Coleman *et al.* [21].

A *state* in a UML statechart diagram is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event [95]. In a system, objects stimulate each other causing state changes by sending and receiving events. When a specified event occurs and the associated guard conditions are satisfied, an object can change its state. Such a state change is called a *transition*. A statechart diagram thus relates events and states.

UML statechart diagrams are drawn as directed graphs in which nodes represent states and directed edges represent transitions. A state is drawn as a rounded rectangle containing the activities it performs in that state and an optional name for the state, separated with a horizontal line from the action part. A transition is drawn as an arrow from the source state to the target state. Statechart diagrams may also have special kinds of states. An *initial state* indicates the starting point of a statechart diagram. Reaching a *final state* means that the execution of the statechart diagram has completed. There can be only one initial state but several final states in a statechart diagram. An initial state is drawn as a small filled black circle and a final state as a bull's-eye icon.

A state may contain *actions* and *activities*. Actions are atomic and non-interruptible, while activities take time to complete and can be interrupted by an event. An ongoing activity can be expressed as a *nested statechart diagram*, or by a pair of actions: an *entry action* starts the activity and an *exit action* stops it. Entry and exit actions can be individual actions as well. Entry actions are executed when entering the state and exit actions when leaving it. Keywords "do/", "entry/", and "exit/" are attached to activities, entry actions, and exit actions, respectively. A state can also have *internal transitions* that may have actions attached to them. An internal transition is fired when a specified event occurs. That causes the execution of an action attached to it, but not a state change nor an interruption of the activities of the state. The event part is separated form the action part by a slash.

A *simple transition* consists of four parts: an event name, event parameters, a guard condition, and actions. The first three define the circumstances under which the transition may fire. When fired, the actions attached to the transition are executed. A transition without an explicit trigger event is called *a completion transition*. It is fired when the activities of its source state are completed provided that its optional guard condition is satisfied. A *concurrent transition* may have multiple source and/or target states. It represents a synchronization and/or a splitting of control into concurrent threads [85]. An *action-expression* is a chain of actions, separated with a delimiter. An action-expression must be an atomic, non-interruptible operation. Such an action-expression can be attached to transitions, entry actions, exit actions, or internal transitions. The statechart diagram in Figure 3.6 contains a simple transition with a label and an action, completion transitions, concurrent transitions, an entry action, and and activity.

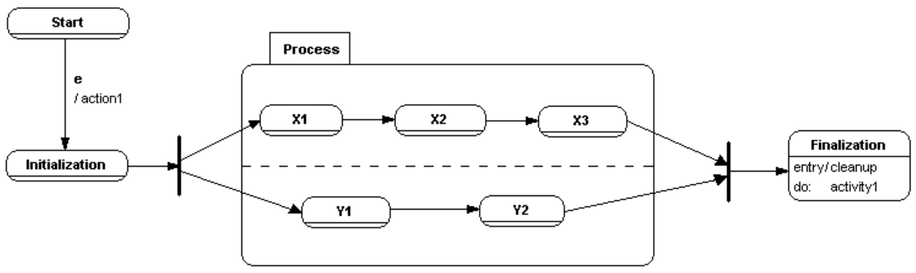


Figure 3.6: A statechart diagram with simple and concurrent transitions. Action *action1* is executed when transition *e* is fired. Entering state *Finalization* , in turn, causes an entry action *cleanup* to be executed, after which an activity *activity1* is started.

Flat state transition diagrams have often been criticized for being impractical and ineffective for modeling large systems. Harel introduces some concepts for raising the expressive power of statecharts [40]. One of them is a *superstate* notation; a way to cluster and refine states. The semantics of a superstate is an *exclusive-or* (XOR) of its *substates*; to be in a superstate an object must be in exactly one of its substates. A superstate is drawn as a large rounded box enclosing all of its substates. Transitions drawn to enter a superstate contour are entering the initial state enclosed inside the superstate. Transitions drawn to leave the superstate contour apply to any of its substates. An example of state generalization by using the superstate notation is shown in Figures 3.7 and 3.8. The information content of these state diagrams is the same (i.e., exactly same paths/event sequences can be found from them). Even these two small state diagrams give an idea of the benefits of nesting states; it is a powerful tool against combinatorial explosion of transitions. UML statecharts can be nested similarly. The contents of the superstate, which is called *a composite state* in UML, can also be shown as a separate statechart diagram. In that case, running the enclosed statechart diagram is expressed as an activity. Collapsing the contents of statechart diagrams this way provides extended means to keep the diagrams compact. Figure 3.9 shows such a high-level statechart diagram constructed from the state machine in Figure 3.8.

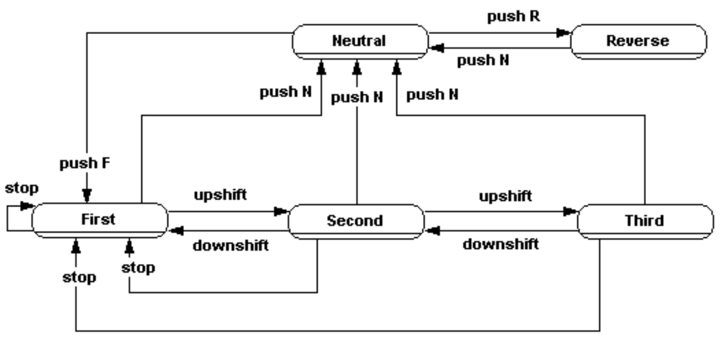


Figure 3.7: A flat state diagram of a car transmission

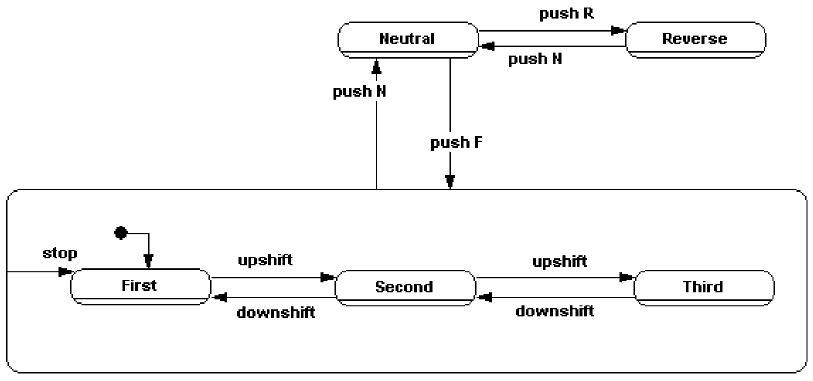


Figure 3.8: A nested state diagram of the car transmission

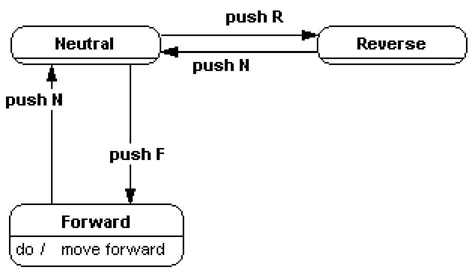


Figure 3.9: A high-level statechart diagram of the car transmission

In addition to XOR composition of states, there is another way of structuring Harel's statecharts, namely AND decomposition. A state consisting of AND components is said to be an *orthogonal product of its components*, meaning that being in such a state, the system must be in all of its AND component subcharts. Hence, orthogonality is a way of representing independence and concurrency. Such a state is visually shown as a state box that is split into components using dashed lines, each separated component representing one orthogonal component [40]. Transitions drawn to enter a superstate contour are interpreted to enter the initial states of all enclosed orthogonal statecharts. A transition drawn to leave the superstate contour applies to all the substates the system is currently in. Figures 3.10 and 3.11 show two Harel's statecharts both capturing the same information; Figure 3.10 shows a flat statechart and in Figure 3.11 there is a state diagram with two orthogonal components. The label *b(in G)* of the transition from state *B* to state *A* in Figure 3.11 expresses conditionality: the transition is fired when an event *b* occurs but only if the system is in state *G* of the orthogonal component. Similar notation is used in UML statechart diagrams for expressing concurrency. If there is a completion transition drawn to leave from the contour of such a decomposed state, the state is exited when all the subdiagrams have reached their final state.

In addition to UML statechart diagram concepts discussed in this chapter, the notation includes *history states*, *deep history states*, *junction states*, *joins*, and *submachine reference states*. These concepts are not reviewed in this dissertation.

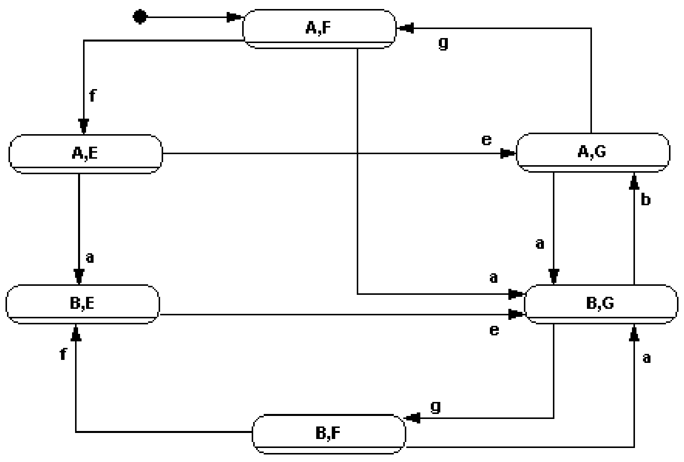


Figure 3.10: A flat statechart

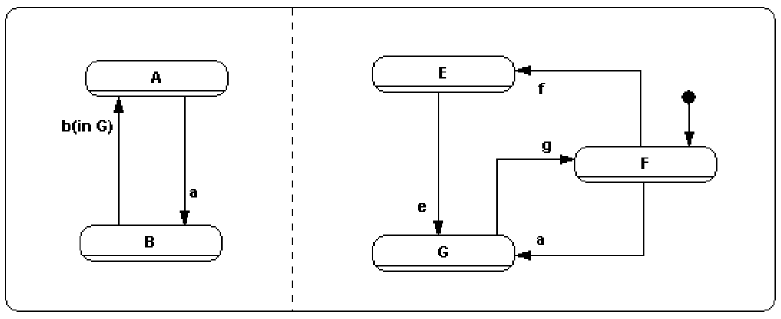


Figure 3.11: A statechart that consists of two orthogonal components

3.5 Activity diagrams

An activity diagram is a special case of a statechart diagram, in which the states are primarily *activity states* or *action states* and in which all (or at least most) of the transitions are triggered by the completion of an activity in the source states [95](p. 135). An activity state has an internal computation and at least one leaving completion transition that fires on the completion of the activity of the state. It should not have any internal or outgoing transitions that are based on explicit events. An action state is atomic, i.e., it cannot be interrupted by transitions. In addition, a way to nest activity diagrams using *subactivity states* is introduced in UML [85]. A subactivity state invokes another activity diagram. The subactivity state is not exited until the final state of the nested diagram is reached, or when trigger events occur on transitions coming out of the subactivity state. An activity diagram suites best for describing situations where all or most of the events represent the completion of internally generated actions (i.e., for procedural flow of control). An activity state is drawn as a shape with straight top and bottom and with convex arcs on the two sides. There is no special notation for an action state. It is usually drawn as a ordinary state. Figure 3.12 shows an example of an activity diagram.

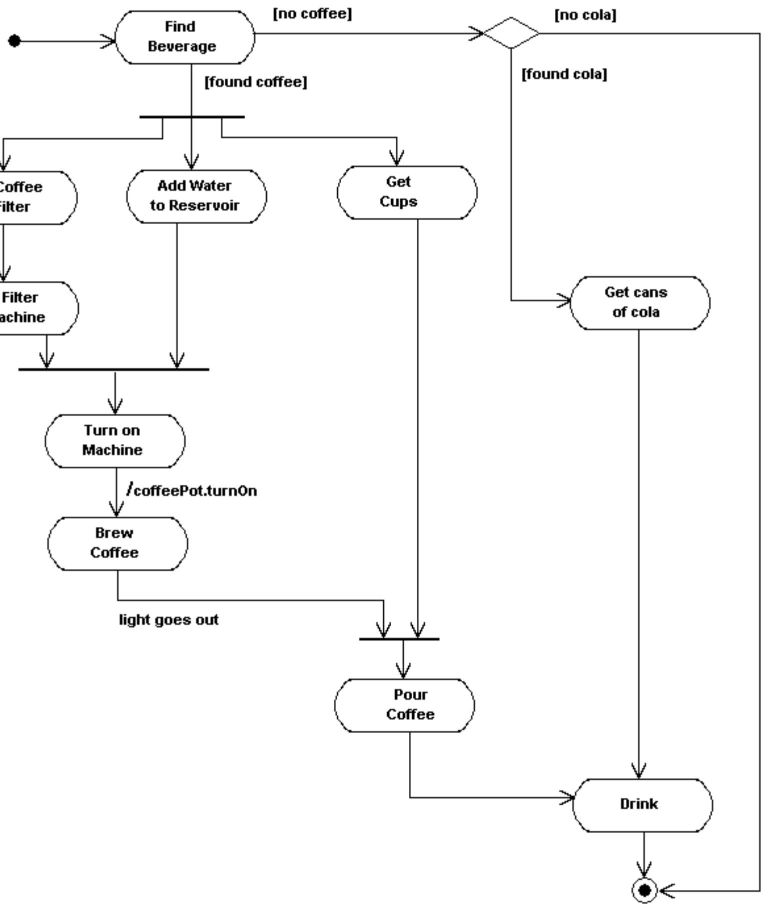


Figure 3.12: An activity diagram[85]

Chapter 4

SCED

SCED has been developed to improve automated support for dynamic modeling in OO software construction [56]. During the development of SCED, the OMT methodology [94, 90, 91] was chosen as a guideline and notation basis. The diagrams used for dynamic modeling in OMT are extended in UML. SCED diagrams are also extended from OMT ones, providing notations that are semantically close to corresponding diagrams in UML. The graphical notation of the diagrams differs, the reason being that SCED was developed during years the 1992 – 1995, while the earliest version of UML, called Unified Method, was introduced in October 1995.

The name "SCED" comes from words SCenario EDitor, since the scenario editor part was implemented first. This emphasizes one of the main ideas behind SCED: the dynamic modeling starts with constructing scenarios. When a sufficiently complete set of scenarios exists, they are transformed into a state diagram for desired participating objects. SCED was developed to be used in forward engineering of object-oriented software systems, although the tool is usable also for other kinds of modeling tasks with a scenario driven approach. In this dissertation, it is shown how SCED can be used for reverse engineering purposes.

An overview of the implementation is presented next. The rest of this chapter introduces the scenario and state diagram notations of SCED. The content of this dissertation relies heavily on the proposed notations. For detailed information about SCED, the reader is referred to [55, 56, 57, 67, 68, 69, 110].

SCED consists of two independent editors: a scenario editor and a state diagram editor. Most of the user interaction is concentrated in these editors. The third part of SCED is a generator, which is just a command activated part (i.e., it is not visible to the user). At any time during scenario editing the user can select one participating object and ask the generator to synthesize a state diagram automatically for it. The synthesis can be done for one scenario only or for a specified set of scenarios. In the latter case, the generator synthesizes each scenario that includes the selected object. Moreover, scenarios can be synthesized to an existing state diagram. The resulting state diagram is then editable using the state diagram editor. The state diagram synthesis algorithm is discussed in Chapter 5 in more detail. The generator can also be asked to optimize a state diagram by adding UML statechart diagram components to it. The optimization algorithms are described in Chapter 6. Finally, some support for checking the consistency between a state diagram and scenarios is available, as discussed in Section 4.2.

SCED was developed in and for the Microsoft Windows environment. The software can be run under Windows NT and Windows 95/98. The tools that are being used for the development work have been selected so that porting to Unix with OSF/Motif should be possible with moderate effort. These tools, versions of which have been changed several times during the development of SCED, are:

1. Borland C++ — C++ language compiler [31, 108, 109].
2. LEDA — *Library of Efficient Data types and Algorithms* [75]. Portable across wide range of platforms.
3. wxWindows — GUI library [101]. Portable between MS–Windows, Windows NT, Motif, Open Look.

4.1 Dynamic modeling using SCED

SCED offers two diagrams for dynamic modeling: a scenario diagram and a state diagram. Scenarios and state diagrams are editable. The editors are typical direct manipulation editors with a graphical user interface. In Sections 4.1.1 and 4.1.2 scenario diagram and state diagram notations of SCED are presented, respectively. In addition, the differences between the SCED scenario diagram notation and the UML sequence diagram notation are discussed. Similarly, the SCED state diagram notation is compared with the UML statechart diagram.

4.1.1 Scenario diagrams

A scenario diagram in SCED corresponds to a sequence diagram UML. In SCED participating objects/classes of a scenario are called *participants*. A participant is drawn as a vertical line and its name is written above the line. Events sent from one participant to another are drawn as horizontal arcs between participant lines. As in UML, the basic MSC notation has been extended in various ways in SCED; additional constructs are allowed in order to make scenarios more expressive, compact, readable, and precise as depicted in Figure 4.1. Such constructs are:

1. an action box,
2. an assertion box,
3. a state box,
4. a comment box,
5. a conditional construct,
6. a repetition construct, and
7. a subscenario.

When modeling an object interaction, it is often necessary to present also other actions than events. At some point, an object may perform arbitrary computations without sending a message. For such actions an *action box* can be placed at a desired vertical position, attached to the owner participant. An action box can also be used to show an event for which the receiver remains undefined, or an event that is received by the sender itself. Action boxes are drawn as rectangles.

Conditions that are known to hold for a particular object at certain positions in a scenario can be expressed using an *assertion box*. The notation of an assertion box follows the CCITT scenario notation standard [13] (currently called ITU [49]). Corresponding to an action box, an assertion box can also be used to denote an event for which the sender remains undefined. The generator interprets an action box like a sent event and an assertion box like a received event when synthesizing a state diagram for the participant. A third kind of box associated with a single participant is a *state box*. A state box gives a name to a particular situation in a scenario from the point of view of a certain participant (i.e., the name of the state of an object at that situation). In addition to a design aid, state boxes can also be used for technical reasons to guide the state diagram synthesis process. The principles concerning the use of these state names during the synthesis process is described in Chapter 5 in more detail. State boxes are drawn as stretched hexagons (cf. Figure 4.1).

In forward engineering, conditions in assertion boxes might be given (e.g., in terms of the attribute values of an object). The usage of state boxes is sometimes convenient for the designer to express her assumption that an object should be in an identifiable state in a particular time position of a scenario. When applying SCED for dynamic reverse engineering of Java software state boxes are used to identify branching points in the execution, that is, locations of conditional structures (e.g., an **if** statement) in the software (cf. Section 8.4.2). Assertion boxes are used to express the path taken as a result of testing such a condition (e.g., whether the condition in the **if** statement yielded true or false). Action boxes are used to indicate an internal method call of an object.

Action, assertion, and state boxes are all attached to one participant only. Next we introduce constructs that concern several participants at the same time. First, a *comment box* can be stretched over (and concerning) several participants. Comment boxes are drawn as rounded rectangles including editable plain text. Comments have no effect on the synthesis process (i.e., a comment

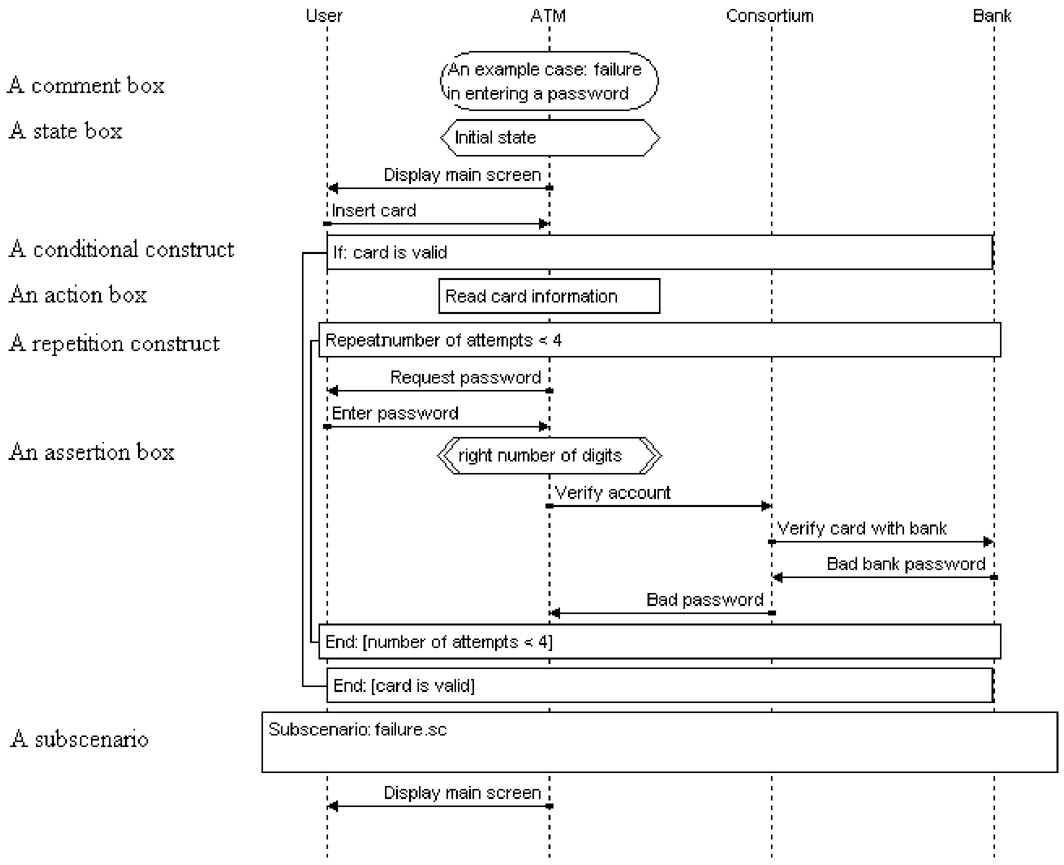


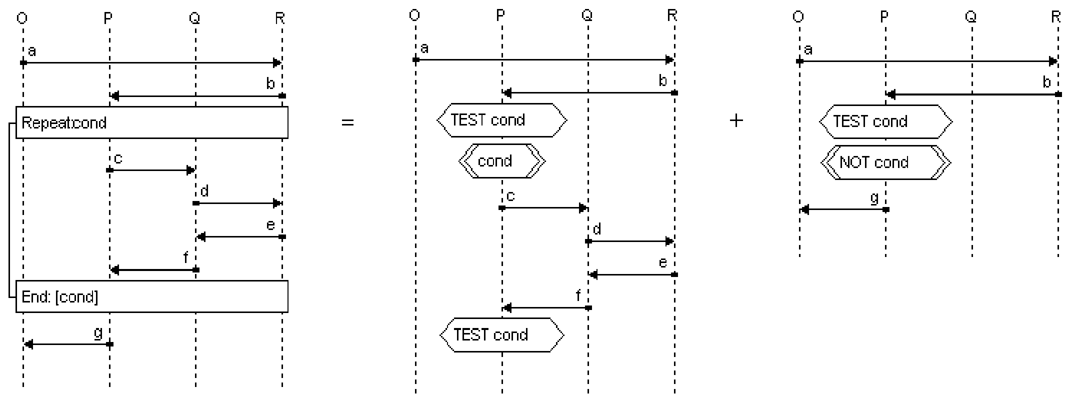
Figure 4.1: SCED scenario diagram notation

written in a scenario cannot be seen in a state diagram after the synthesis process).

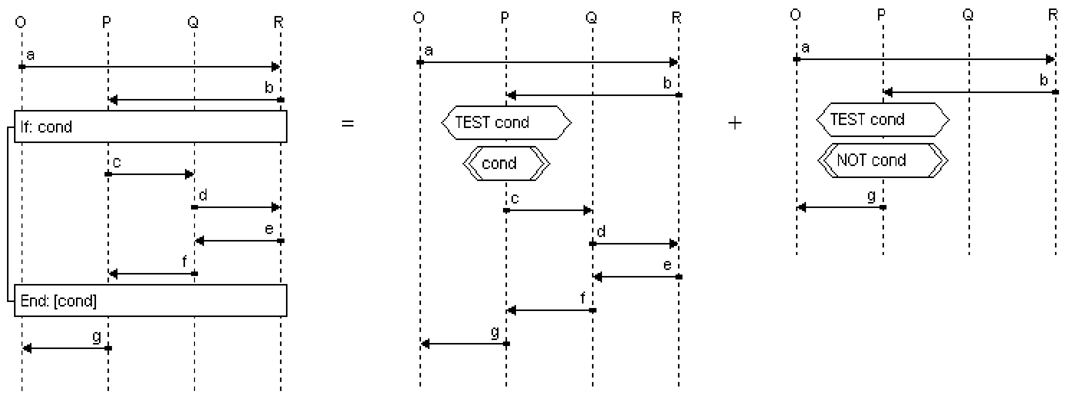
To support the presentation of a use case as a single scenario diagram, the scenario notation in SCED has been extended with algorithmic constructs to express conditionality and repetition. A certain object can, however, be involved in several use cases. Hence, such algorithmic scenarios are full specifications for use cases but (usually) not for objects. With a conditional construct the designer may separate sequences of events that occur only under certain circumstances. A conditional construct consists of two rectangles and a bent line connecting them. A keyword “if” is associated with the first rectangle specifying the beginning of the construct. The designer specifies the condition after the word “if”. A keyword “end” is associated with the end part of the construct, after which the condition in brackets is automatically inserted. A repetition construct is drawn like a conditional construct, the only difference being that the keyword “repeat” is used instead of “if”. The graphical notation of conditional and repetition constructs is shown in Figure 4.1.

Algorithmic scenarios can be interpreted as sets of ordinary scenarios. The interpretation is shown in Figure 4.2. Note that in the case of repetition the number of iterations (and therefore the number of scenarios) is potentially infinite, but the repetition construct can nevertheless be represented making use of the state box; after evaluating the loop expression and executing the body, the participant will be in the same state as before entering the loop. Hence, the participant will be able to re-execute the loop infinitely. The synthesizer reads algorithmic constructs recursively, since they can be arbitrarily nested (cf. Section 5.2).

Analogously to subroutines, a scenario may consist of parts that have their own aims and characterizations. For instance, a scenario describing the usage of an ATM might include event sequences like “checking a card” or “giving a correct password”, for which one can construct separate scenario diagrams and then just “call” these scenarios using a *subscenario box* instead of repeating their contents. The subscenario box notation has been adopted to SCED in order to make scenarios easier to read and write, and to simplify and to structure them. A subscenario box is drawn as a rectangle stretched over all participants. It can be placed at any vertical position in a scenario. Af-



Expanding a repetition construct



Expanding a conditional construct

Figure 4.2: Dissolving conditional and repetition constructs into simple scenarios

ter creating a subscenario, a key word “Subscenario:” appears in the upper left corner of the box, after which the designer can write the file name of the subscenario. Semantically, a subscenario box is a shorthand notation for including the contents of the referred scenario to the position of the subscenario box.

The participants of the subscenario can be different from those of the host scenario. Hence, a subscenario box is especially useful if the subscenario requires objects that are not needed for the rest of the scenario: the host scenario becomes smaller both in vertical (event sequence) and in horizontal (participants) direction. Structuring scenarios using subscenario boxes helps the user to understand the contents of the scenario set. In forward engineering, the designer can divide scenarios into logical units by using subscenarios that have clear, intuitive meaning. These units can then be modified (refined) afterwards separately, if needed. In dynamic reverse engineering, subscenarios have been used to view behavioral patterns found by string matching algorithms (cf. Section 8.5). If the pattern is repeated several times in a row, a repetition construct is used instead. The generator handles subscenarios recursively, since subscenarios can naturally include other subscenarios.

The SCED scenario diagram notation is not as rich as the UML sequence diagram notation (cf. Figures 3.3 and 3.4). For example, there is no way to express object deletion in SCED scenario diagrams. It can only be said that a certain participant is no longer needed, if after a certain point there are no more scenario items attached to it. Furthermore, there is no corresponding notation to recursion nor activation of a participant. The creation and the deletion of an object can be expressed using an event with an appropriate label. For example, when applied for reverse engineering of Java software (cf. Section 9.3.5), an event of type $< init > (\ldots)$ is generated when an object is created (i.e., the invocation of a constructor). Recursion can be expressed using a repetition construct (or in some cases a subscenario). Fork of control, in turn, can be expressed using a conditional construct or assertion boxes. However, an else-part of a conditional construct is not implemented in SCED. To express an **if-else** structure the designer needs to construct two scenarios: one with a conditional construct expressing the **if** part and another expressing the **else** part. Table 4.1 enumerates the UML sequence diagram constructs and either the corresponding SCED scenario diagram construct or constructs that can be used in SCED scenario diagrams for expressing the meaning of the UML sequence diagram construct in question.

A UML sequence diagram construct	A SCED scenario diagram construct	Correspondence
synchronous message	synchronous event	full
asynchronous message	asynchronous event	full
return from procedure call		missing from SCED
self event	an action box	full
object	participant	full
activation of an object		missing from SCED
conditional branching	conditional construct	partially replacing
creation of an object	an event with a specific label	replacing
deletion of an object	an event with a specific label	replacing
recursion		missing from SCED
marked iteration	repetition construct	can be replaced in UML
a comment	a comment box	full
a constraint	an assertion box	partially replacing
	a subscenario box	missing from UML

Table 4.1: UML sequence diagram constructs and either the corresponding SCED scenario diagram constructs or constructs that can be used in SCED scenario diagrams for the task in question. The correspondence is characterized as “full”in the former case, “replacing” in the latter case, and “missing” if either of the notations contains constructs that cannot be expressed with the other.

In addition to UML and SCED, restricted forms of algorithmic scenario notations have been applied by Portner [78] and De Pauw *et al.* [27]). Notation constructs that correspond to SCED subscenarios and assertions are included, for example, in the Life Sequence Charts (LSC) notation [24]).

4.1.2 State diagrams

The SCED state diagram notation does not contain all the notation constructs included in UML. Furthermore, some of the constructs have slightly different meaning.

A *state* in a SCED state diagram represents a particular point reached in a computation. The definition of a state in UML includes this interpretation: a state in a UML statechart diagram is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event [95]. A commonly used definition of a state characterizes a state as an abstraction of a particular combination of the attribute values and links of the object. This definition is used by OMT [94]. The somewhat vague interpretation of a state in UML accepts this definition, but not the interpretation used in SCED. There are several reasons for that. First, by states and transitions in a SCED state diagram we indeed intend to describe how the object can get to a certain state and to which events it may respond while being in that state. Second, as described in Chapter 5, states of a participating object are defined by events the object sends and is able to receive, not by its attribute values and links. Third, by adding advanced UML constructs to a state diagram, the size of the state diagram can be decreased in terms of the number of states, as described in Chapter 6. This is basically done by removing states that are entered automatically without receiving any external stimulus. Again, only an interpretation of a state that is based on events allows this.

The SCED state diagram notation does not include activities that can be interrupted by an event. State diagrams are (usually) generated automatically on the basis of information given in scenarios (cf. Chapter 5); actions and transitions are named according to events appearing in scenarios. When synthesizing a state diagram, the generator has no way to conclude which events correspond to instantaneous actions and which ones to continuous or sequential activities. The question is highly semantical and there is no means to specify the duration of an event in SCED scenario diagrams. Nonetheless, if interruptible events and activities were included in SCED scenario and state diagram notations, hardly any changes would be necessary to either state diagram synthesis or the following optimization algorithms. Only few additional constraints would be needed, as described in Section 5.2 and in Chapter 6.

Several normal actions (denoted with a keyword “do:”) placed in a single state in a SCED state diagram are executed in succession. Furthermore, all the actions, except possibly the last one, are executed without an interruption by any event. There are two reasons for this interpretation. First, the information of a synthesized state diagram is read directly from scenarios that contain only sequential information. Second, the execution of the actions is not interruptible due to the way actions can be “packed” into a single state (cf. Section 6.2).

Expressions and interpretations of entry actions, exit actions, and actions attached to transitions are similar in SCED and UML. Furthermore, they may all include an action-expression instead of a single action the same way as in UML. The delimiter used in SCED state diagrams is a comma. An internal transition in UML, however, differs slightly from the interpretation of an *internal action* in SCED. In UML, an internal transition *e/act* of a state *s* is interpreted as follows: if event *e* occurs, action *act* is executed but neither entry- nor exit actions of *s*. This applies to a corresponding internal action in SCED as well. In addition, in UML the execution of possible activities of *s* is not interrupted by the execution of *act*. A following interpretation is used in SCED instead: all normal actions of *s* are executed (in a row) after the execution of *act*. Reasons for these interpretations are discussed in Sections 6.2 and 6.4 in more detail.

In UML, states may have substates. Adopting the composite state construct to SCED state diagrams is desirable because of its ability to outline the structure of state diagrams and to decrease the number of transitions needed. In SCED, a combination of *superstates* can be generated for a flat state diagram [110]. Superstates may have entry and exit actions but not normal nor internal actions. The layout algorithm of SCED does not currently show the superstates generated.

Perhaps the most fundamental difference between UML statecharts and SCED state diagrams is the fact that SCED state diagrams do not have a notation for concurrency. Interruptible activities are not supported by SCED. This, together with the state diagram synthesis and state diagram optimization algorithms, results in a state diagram in which states are defined either by a single action or by several actions that are executed in succession. Particularly, an object is always in a single state. In UML state diagrams, a state is a product of states of its concurrent components, if any exist. In order to be able to synthesize states that have concurrent components in SCED, there should be a way to link scenarios or participants so that states could be able to recognize their aggregates.

The main UML statechart diagram constructs have been presented in Section 3.4. Most of those constructs are supported by SCED. Table 4.2 enumerates UML statechart diagram constructs and the corresponding SCED state diagram constructs.

A UML statechart diagram construct	A SCED scenario diagram construct	Correspondence
a normal state	a normal state	partial
a state name	a state name	full
an initial state	an initial state	full
a final state	a final state	full
a transition	a transition	full
activity		missing from SCED
	normal action	missing from UML
an action-expression	an action-expression	full
an entry action	an entry action	full
an exit action	an exit action	full
actions attached to transitions	actions attached to transitions	full
an internal transition	an internal action	partial
a guard condition	a guard	full
nested states	nested states	partial
concurrent substates		missing from SCED
constraints		missing from SCED
a history state		missing from SCED
a deep history state		missing from SCED
a junction state		missing from SCED
a submachine reference state		missing from SCED
a join		missing from SCED

Table 4.2: UML statechart diagram constructs and the corresponding constructs of SCED state diagrams. The correspondence is characterized as “full”, “partial”, or “missing”, an in Table 4.1.

4.2 Examining the models

SCED provides tools that can be used to examine the structure of a state diagram in various ways, tools for checking consistency between scenario diagrams and state diagrams, and tools for changing a state diagram on the basis of scenario diagrams synthesized to it. Such an automated support is useful since both of the diagram types are editable.

Several tools have been implemented to SCED for examining various structural aspects of a state diagram. One of them can be used for finding states that are non-deterministic. A state is non-deterministic if two similarly labeled leaving transitions enter different states. Non-determinism also occurs if the event triggering an internal action is the same as a label of a leaving transition. In these cases there is no unique interpretation or response to a received event. SCED also implements algorithms that can be used to detect reachability of states. In a properly constructed state diagram there may only be one initial state but several final states. An initial state as well as final states are added by the designer. For helping the designer to add final states to the state diagram, SCED provides a tool for finding all states from which no event may cause a state change (i.e., states that have no leaving transition attached to them). The final states should probably be entered from these states. If not, the algorithm at least points out the parts of the state diagram that need further revision. Finally, an algorithm can be used for finding states that can never be reached from the initial state.

Some automated tools are offered for helping the designer to detect inconsistencies between scenario diagrams and a state diagram. For example, a tool is provided for running a scenario against a state diagram. The corresponding path is highlighted in the state diagram. If a scenario is no longer accepted by the state diagram, another tool can be used for showing the exact point in which the scenario becomes invalid.

SCED provides various tools for changing a state diagram on the basis of scenarios synthesized to it. As a counterbalance to the incremental state diagram synthesis, SCED offers a tool for desynthesizing scenarios out of the state diagram. Just as the synthesis algorithm, the desynthesis algorithm leaves user editions untouched. Furthermore, a state can be split into two or more states so that the resulting state diagram is still consistent with scenarios. This tool is provided since the synthesizing algorithm is eager to join states, which may result in an undesirable state join. An equally important feature to a split operation is a merge operation. The user may merge several states into a single one if all of the states are exactly similar, that is, they have the same actions and state names. The merge may yield to a nondeterministic state. Such a situation could be handled in two different ways: the merge could be denied or distinguishing guard conditions could be added to identically labeled leaving transitions in order to avoid the nondeterminism. In SCED the latter principle is supported.

4.3 Summary

SCED is a dynamic modeling tool that uses OMT as a notation basis. The rather rudimentary OMT scenario diagram notation has been extended in various ways in SCED. The extended SCED scenario diagram notation is semantically similar to the UML sequence diagram notation [95, 85] but it follows the CCITT standard [13] (currently called ITU [49]). The SCED scenario diagram notation lacks some UML sequence diagram constructs. Most of them can be replaced by other SCED scenario diagram constructs. Scenario diagrams can be nested in SCED using subscenario boxes. Subscenario boxes are powerful constructs to emphasize behavioral patterns, hence simplifying and structuring the scenarios. Such a construct is lacking from UML sequence diagrams. A subscenario construct is also adopted by other tools (e.g., TED [121]).

The most significant difference between UML statecharts and SCED state diagrams is the fact that SCED state diagrams do not have a notation for concurrency. Some other UML statechart diagram constructs are missing from SCED state diagrams as well. SCED does not support other behavioral UML models.

In this dissertation, SCED is used to support dynamic reverse engineering of Java software systems. The dynamic event trace information is visualized as SCED scenario diagrams. The object interaction of both single and multi-threaded programs can be shown as scenario diagrams. The overall behavior of one object or a method can be depicted as a state diagram, using the state diagram synthesis feature of SCED.

Chapter 5

Automated synthesis of state diagrams

In this chapter we discuss algorithms to synthesize a state diagram automatically from a set of scenario diagrams. This technique allows the engineer to examine the total behavior of an object as a single model, disconnected from the rest of the system. The presented algorithms have been implemented in the SCED tool.

In UML-based behavioral modeling, sequence diagrams are usually given first for “normal” cases, and then for various cases representing “exceptional” behavior. When a sufficiently complete set of sequence diagrams exists, statechart diagrams are constructed for desired participating objects. Not all objects usually need a statechart diagram. For some of them it would not even be meaningful or sensible to form one (e.g., for actors outside the system border). In addition, some objects have such a trivial behavior that constructing a statechart diagram for them would not give any useful information to the designer. Furthermore, a sequence diagram often contains participants that do not even represent objects (e.g., the actual end user).

A sequence diagram and a statechart diagram share common information, both of them describing dynamic aspects of a system. Consequently, a sequence diagram contains information not included in a statechart diagram, and vice versa. While a sequence diagram shows an example of communication among objects, a statechart diagram describes the total behavior of a single object.

A basic observation behind SCED is that the synthesis of a state diagram on the basis of given scenarios can be carried out automatically. The basic synthesis algorithm used has been presented by Biermann and Krishnaswamy [7], and its adoption to state machine synthesis from scenarios is discussed by Koskimies and Mäkinen [54]. The Biermann-Krishnaswamy algorithm (hereafter BK-algorithm, for short) and the way it is applied in SCED are presented next. The notation and principles used closely follow those in [7] and [54]. Finally, some criticism and remarks concerning the state diagram synthesis are presented.

5.1 The BK-algorithm

Biermann and Krisnashwamy present their algorithm (BK-algorithm)for synthesizing programs from their traces [7]. The idea is that the user specifies the data structures of a program and describes (graphically) its expected behavior in the case of an example input in terms of primitive actions (like assignments) and conditions that hold before certain actions. Essentially, the user gives traces (i.e., sequences of actions and conditions) of the expected program, and the algorithm produces the smallest program that is capable of executing the given example traces. Moreover, after giving some finite number of example traces taken from a program, the algorithm produces a program that can execute exactly the same set of traces as the original one, that is, the algorithm learns (or infers) an unknown program. Before discussing the principles of the BK-algorithm and studying its properties, it is necessary to introduce the notation used and definitions related to it.

A computation consists of condition-instruction pairs $r_t = (c_t, i_t)$ being executed at discrete times $t = 1, 2, \ldots$. Instruction i_t operates on memory contents m_{t-1} resulting in a new memory contents m_t; this is denoted by $m_t = i_t(m_{t-1})$. Branching is made possible by using conditions that are (possibly empty) conjunctions of atomic predicates or their negations. The value of condition c_t on memory contents m_{t-1} is denoted by $c_t(m_{t-1})$. The value of the empty condition ϕ is true. The validity of condition c_t is checked before instruction i_t is executed.

Instructions available are denoted by $I_0, I_1, I_2, \ldots, I_z$, and I_H, where I_0 denotes the start instruction and I_H is the halt instruction. Every program has exactly one occurrence of I_0 and usually one occurrence of I_H. An instruction may appear several times during a computation. The appearances are separated by labels. For example, the appearances of I_1 are labeled by $1I_1, 2I_1, 3I_1, \dots$.

A *partial trace T* of a computation is a $(2n+1)$-tuple

$$T = (m_0, r_1, m_1, r_2, m_2, \dots, r_n, m_n),$$

where, for each $t = 1, 2, 3, \dots, n$, we have

r_t is a condition-instruction pair $r_t = (c_t, i_t)$,
$m_t = i_t(m_{t-1})$,
$c_t(m_{t-1})$ is $true$, and $c_1 = \phi$.

A trace is a partial trace $T = (m_0, r_1, m_1, r_1, \dots, r_n, m_n)$ with the additional requirements that $r_1 = (\phi, I_0)$ and $r_n = (c_n, I_H)$. An *incomplete program P* is a finite set of triples of the form (q_j, c_k, q_e), where each q_j and q_e is a labeled instruction and c_k is a condition, and where the following restriction holds:

If $(q, c, q') \in P$ and $(q, c', q'') \in P$ and there exists m such that $c(m) = c'(m) = true$, then $c = c'$ and $q' = q''$.

Thus, an incomplete program is a finite set of labeled instructions connected by triples or *transitions* which are all associated with a particular condition. A transition is executed if its condition is true. Moreover, an incomplete program is deterministic (i.e., no two applicable transitions can ever be simultaneously satisfied).

A *program* is an incomplete program with the additional requirements that

1. Each program has exactly one start instruction, namely $1I_0$,
   and $(1I_0, c, q) \in P$, for some c and q.

2. If there is a transition (q, c, p), where c is a non-empty condition, then there must also be transitions from q for every possible combinations of the atomic predicates and their negations presented in c.

Programs can be illustrated as directed graphs with instructions as nodes and transitions as edges. Edges are labelled with conditions. So, empty conditions correspond to unlabelled edges. Figure 5.1 illustrates a program with transitions

$(1I_0, \phi, 1I_1), (1I_1, \{a\}, 2I_1), (2I_1, \phi, 1I_2), (1I_2, \phi, 1I_1)$, and $(1I_1, \{\neg a\}, 1I_H)$.

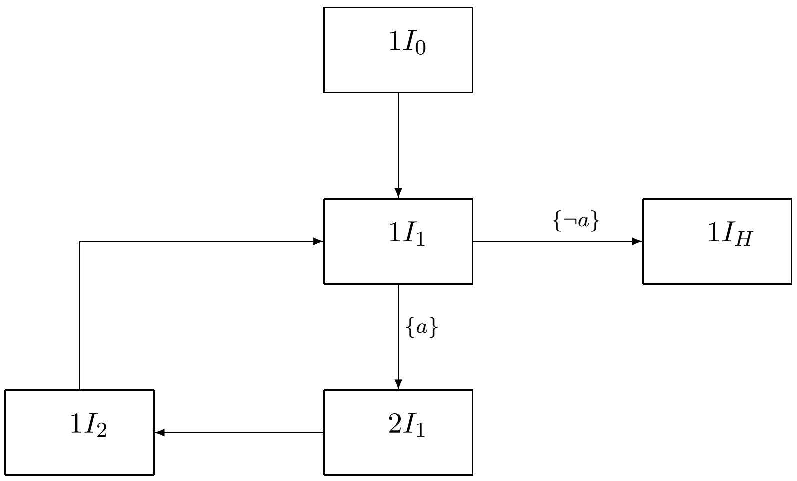


Figure 5.1: Program $P = \{(1I_0, \phi, 1I_1), (1I_1, \{a\}, 2I_1), (2I_1, \phi, 1I_2), (1I_2, \phi, 1I_1), (1I_1, \{\neg a\}, 1I_H)\}$.

A Java version of the program in Figure 5.1 could be written as shown in Figure 5.2.

The input of the BK-algorithm is a set of traces from which the algorithm infers a program consistent with the traces. The algorithm defines a minimum labeling for instructions given in these input traces. This means that the number of different instances of an instruction in the resulting program, and hence the number of nodes in the directed graph illustrating the resulting program, are minimized. Consider two labelings of n instructions, say $U = (u_1, u_2, \ldots, u_n)$ and

```
public class X {
        . . .
        public void exampleMethod() {
                I0();
                I1();
                while (a) {
                        I1();
                        I2();
                        I1();
                } IH();
        }

. . .
```

Figure 5.2: A Java version of the program in Figure 5.1.

$V = (v_1, v_2, \ldots, v_n)$. We denote $U < V$, if there is j, $1 \leq j \leq n$, such that $u_i = v_i$, for $i = 1, \ldots, j-1$, and $u_j < v_j$. Let k be the number of different instances of instructions in the corresponding program (i.e., k is the number of nodes in the directed graph that represents the program). The rank of labelings is defined for pairs of the form (k, U). If (k, U) and (k', U') are two such pairs, we denote $(k, U) < (k', U')$ if $k < k'$ or if $k = k'$ and $U < U'$. Hence, one starts with a pair $(1, (1, 1, \ldots, 1))$ and enumerates all the pairs (k, labeling) in increasing order, until a pair that defines an incomplete program is found. This process will always halt, since the pair $(n, (1, 2, \ldots, n))$ defines a linear program with no branching or loop.

As an example, consider the trace

$$(m_0, (\phi, I_0), m_1, (\phi, I_1), m_2, (\{a\}, I_1), m_3, (\{a\}, I_2), m_4, (\phi, I_1), m_5, (\{\neg a\}, I_H), m_6).$$

Since the trace contains four different instructions (I_0, I_1, I_2, I_H) it is useless to try values less than $k = 4$. The only possible labeling with $k = 4$ is $(1, 1, 1, 1, 1, 1)$. This labeling would give transitions

$$(1I_0, \phi, 1I_1), (1I_1, \{a\}, 1I_1), (1I_1, \{a\}, 1I_2), \text{ and } (1I_1, \{\neg a\}, 1I_H).$$

This set of transitions contradicts the requirements of determinism (transitions $(1I_1, \{a\}, 1I_1)$ and $(1I_1, \{a\}, 1I_2)$). Hence, we set $k = 5$. Labeling $(1, 1, 1, 1, 2, 1)$ implies nondeterminism as well, but labeling $(1, 1, 2, 1, 1, 1)$ gives the program illustrated in Figure 5.1.

We say that a program P can *execute* trace

$$(m_0, (c_1, i_1), m_1, (c_2, i_2), m_2, \ldots, (c_n, i_n), m_n)$$

if there is a labeling $(u_1, u_2, \ldots, u_n)$ such that $(u_j i_j, c_{j+1}, u_{j+1} i_{j+1})$ is a transition in P and $c_{j+1}(m_j)$ is true, for each $j = 1, 2, \ldots, n$. Biermann and Krishnaswamy [7] have proved the following: given a set of traces, the program inferred by the BK-algorithm, principles of which are given above, can execute all input traces. Moreover, if P is any program whose traces are given as input, then the BK-algorithm identifies P, which has the minimal number of different instances of instructions (i.e., the directed graph illustrating the resulting program has a minimal number of nodes).

5.2 Applying the BK-algorithm to state diagram synthesis

From the point of view of the particular object, similarities between the concepts of a scenario and a trace (the input of the BK-algorithm) can be found. There is also a correspondence between a state diagram and a program (the output of the BK-algorithm). Due to the similarities, the BK-algorithm can also be applied to state diagram synthesis. However, the relation is not trivial. The fundamental difference between programs and state machines is that a program (in the sense of the BK-algorithm) is a self-contained process, which needs no external stimuli, while the whole purpose of a state machine is to describe the behavior of an object in the presence of external input. In other words, programs are more complete than state machines. Hence, for a program, the flow of control is fully determined by the program itself, but for a state machine the flow of control depends on the sequence of events sent by some process that is in principle unknown [54]. The notation and algorithms presented in this section are based on those given by Biermann and

Krisnashwamy [7] and by Koskimies and Mäkinen [54].

First, let us consider the relations between the concepts of a trace in the BK-algorithm and events in UML. In UML, a (received event, sent event) pair acts in the same role as a (condition, instruction) pair in the BK-algorithm. A received event corresponds to an operation call on the object, causing the object to react, that is, it sends an event to some object(s) (possibly also to itself). Similarly, in the notation of the BK-algorithm, if at time t condition c_t is observed to be true, an instruction i_t will be executed. Hence, a program trace corresponds to a vertical object line in a scenario: each sent event corresponds to an instruction, and each received event to a condition. In the resulting state diagram the names of sent events are shown as actions inside states, and names of received events are shown as names of transitions. In SCED, action boxes can be used to represent events that are received by the sender object itself (e.g., calls for object's own operations). Thus, action boxes of the object are treated like sent events.

Figure 5.3 shows two SCED scenario diagrams that represent different example runs through method *exampleMethod* in Figure 5.2. A trace constructed by reading the scenario items of *X* in scenario diagrams *example 1* and *example2* correspond to the trace in Figure 5.1.

With the interpretation above, states will be defined by their actions after the synthesis in SCED. Hence, the state represents a period of time, during which an object possibly sends an event and is waiting to receive one that causes a transition to another state. In UML, a state is defined as a condition or situation during which it satisfies some condition, performs some activity, or waits for some event [95]. The interpretation of a state in SCED thus satisfies this somewhat vague definition of a state in UML. An activity is defined as an execution of substructure within a state machine, that is, a substructure that has duration with possible interruption points [95] (p. 133). An action, in contrast, is defined as an executable atomic computation, which cannot be terminated externally [95] (p. 122). Actions can be associated with several other keywords (cf. Chapter 3). In a synthesized SCED state diagram, however, the keyword "do:" is associated with actions that correspond to sent events in scenarios. It is assumed in SCED that the events often refer to in-

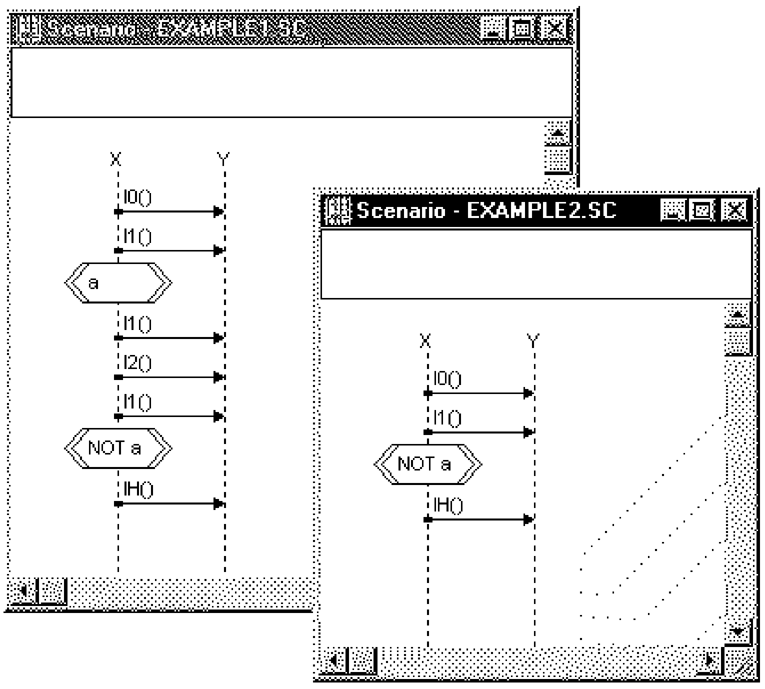


Figure 5.3: Two SCED scenario diagrams that represent different example runs through method *exampleMethod* in Figure 5.2

stantaneous operations, or at least operations that need to be completed without interruption. This assumption is especially justified when SCED is applied for reverse engineering. The scenario diagrams in that case show the method calls between objects. The execution of a method is typically not interrupted, except possibly by a thrown exception.

The meaning of the absence of a condition in BK-algorithm and the absence of a trigger in a transition differ as well. In programs of the BK-algorithm the absence of a condition means that either no test was made or a test was made and it yielded “false” [6] (p. 123). So, if a particular instruction in a program of the BK-algorithm has several leaving transitions, an unlabeled transition and others labeled $C_1, C_2, \ldots, C_k$, an unlabeled transition is fired if none of the conditions $C_1, C_2, \ldots, C_k$ is “true” [6] (p. 123). In a UML statechart diagram, a *completion transition* (i.e. a triggerless transition) without a guard is implicitly triggered on the completion of any internal activity in a state [95] (p. 479). UML allows a completion transition without a guard and a transition with an event trigger to be attached to a same state as leaving transitions, although it is not common practice. However, if the activity in the state was an instantaneous one (i.e., an action), then the labeled transition would never get a chance to fire. In case of an activity that takes time to complete, the transition with the event trigger fires only if the object receives the event before completing the activity.

In SCED, the duration of actions is not defined. Suppose that there are two consecutive sent events, say a and b, in a scenario diagram. The synthesized state diagram will then have one state (say $s1$) with action a, another state (say $s2$) with action b, and an unlabeled transition from $s1$ to $s2$. It is assumed in SCED that a represents an instantaneous action, or at least, is completed without interruption of any event. If we allowed $s1$ to also have a labeled transition as a leaving transition, such assumptions about the duration of a could not be made. In SCED the interpretation of an unlabeled transition is closer to a completion transition of UML than an unlabeled transition of the BK-algorithm: an unlabeled transition in SCED is interpreted to be fired immediately after the action of its source state is completed. In contrast to UML, the synthesis algorithm does not allow a state to have both unlabeled and labeled transitions as leaving transitions. To emphasize the difference of the interpretation of the absence of a label of a transition between SCED and UML, unlabeled transitions are called *automatic transitions* in SCED.

Since transitions may also have guards, the restriction for a state join during the synthesis was modified to the following form: a state cannot have both an automatic transition and a labeled transition as leaving transitions if these transitions have the same guards or they are both unguarded. The algorithm processes guards as strings. Hence, only guards that are written exactly the same way are considered to be the same. This means that there is no way in SCED to guarantee that exactly one transition may fire while being in a certain state; not only because there is no way to inspect the semantic meaning of guards but also because synthesized state diagrams are not complete as explained above. This contradicts a rule used in the BK-algorithm, which requires that there always must be exactly one transition condition satisfied until the halt is reached [7].

A concept corresponding to a trace in the BK-algorithm is defined next. Let *S* be a SCED scenario that has an instance of class *C* as a participating object. A special event called *void* is defined to represent an event that never occurs. Event *void* will not be seen in a state diagram but it is needed in order to handle the end of the trace correctly. A *trace* is a sequence of triples $(e1, e2, s)$, where $e1$ is either empty or a label of a scenario element in *S*, $e2$ is *void*, empty, or a label of a scenario element in *S*, and s is an additional string (empty or a label of a state box in *S*) defining the name of the state, in which $e1$ may be an action. The trace originating from scenario *S* with respect to *C* is incrementally updated by using the following algorithm. Note that the trace may or may not be empty; the information given by *S* is nevertheless added to the current trace. It is assumed that all scenario items (events, action boxes, state boxes, assertion boxes, if-constructs, repeat-constructs, and subscenario boxes) are in top-down order. A scenario item *p* points to the current scenario item. This concept is needed because of recursive calls for the routine *fillTrace*. When calling *fillTrace* for a whole scenario so that all scenario items (from the beginning) will be added, scenario item *p* points to NULL.

Algorithm 1. Filling a trace.
Input: A scenario *S* with an instance of class *C* as a participating object, and a scenario item *p*.
Output: Trace *T* originating from *S* with respect to *C*.
Method:

```
fillTrace(S,p)
  Consider a vertical line corresponding to the instance of class C object.
  Let $S_c$ be a sequentially ordered list of those scenario items in $S$ that concern $C$.
  if $p$ is NULL then
    Let $p$ be the first scenario in $S_c$.
  while $p$ is not NULL do
    Let e (if any) be the name of $p$.
    case $p$ is
      1: a sent event or an action box
  **if** the previous scenario item was a state box **then**
    Let $i = (e1_i, e2_i, s_i)$ be the last item in *T*.
    $e1_i := e$.
  **else**
    Append item $(e, NULL, NULL)$ to *T*.

**2:** a received event or an assertion box
  **if** the previous scenario item was a sent event, an action box, or a state box
  **then**
    Let $i = (e1_i, e2_i, s_i)$ be the last item in *T*.
    $e2_i := e$.
  **else**
    **if** $p$ is an assertion **then**
      Add brackets around $e$ (e.g. *"[eventname]"* ).
    Append item $(NULL, e, NULL)$ to *T*.

**3:** a state box
  Append item $(NULL, NULL, e)$ to *T*.

**4:** a subscenario box
  Find scenario $S'$ corresponding to the label of the subscenario box.
  **if** $S'$ is found **then**
    *fillTrace*$(S', NULL)$.

**5:** the beginning part of an if-construct or a repeat-construct
  // Items of the current scenario $S$ are added twice
  // representing two cases: one for the case where the
  // condition is true (i.e. the items of the construct are
  // read), and another for the case where the condition
```

```
            // is false (i.e. items of the construct are skipped).
            **if** the construct is not empty **then**
                // Read items of the construct:
                Append $i_c = (NULL, "[" + e + "]", "TEST" + e)$ to *T*.
                // '+' indicates a concatenation of strings.
                Get the next scenario item $p'$.
                *fillTrace*$(S, p')$.
                Let $i = (e1_i, e2_i, s_i)$ be the last item in *T*.
                $e2_i := void.$
                // Skip items of the construct:
                Up to $i_c$, append duplicates of those trace items in T that has been added
                during the current call of *fillTrace*.
                Let $i = (e1_i, e2_i, s_i)$ be the last item in *T*.
                Negate the guard of $e2_i$, i.e. add a string "NOT" if it does not already con-
                tain it, otherwise remove "NOT".
                Jump to the item following the end-part of the construct.

        **6:** the end part of an if-construct
            // Do nothing.

        **7:** the end part of a repeat-construct
            Append item $(NULL, "[NOT" + e + "]", "TEST" + e)$ to *T*.
            // It is assumed here that $e$ does not contain brackets, if it
            // does, they are removed first.
    **end** // case
    Let $p$ be the next scenario item in $S_c$, or NULL if all items in $S_c$ are handled.
**end** // while
Let $i = (e1_i, e2_i, s_i)$ be the last item in *T*.
$e2_i := void.$
```

After the execution of routine *fillTrace*, the event part of the last trace item, say e in (f, e, s), is set to be *void*. This is done in order to specify the end of the scenario. Note that this is also done after each recursive call of routine *fillTrace*.

Algorithm 1 can also be used when synthesizing operation descriptions from scenarios (instead of a state diagram). An operation call for an object is shown with an arriving call arc in a scenario. The corresponding return from the operation is shown with a leaving arc. All the leaving arcs between them are internal calls of operations of other objects, and all arriving arcs are returning counterparts of these calls. Hence, the trace of the operation call consists of the internal calls shown by leaving arcs. Operation synthesis does not cause any major extensions to Algorithm 1; only the scenario items to be read vary. In operation synthesis, instead of all scenario items, only items between the operation call and the corresponding return value are read. When calling the *fillTrace* routine the parameter p should point to the operation call. The corresponding return value arc can easily be found (the problem can be considered as a problem of finding the closing parenthesis). The operation synthesis is discussed by Koskimies *et al.* in more detail [57].

From the point of view of a state diagram, a trace item $(e1, e2, s)$ means that the execution has reached a state s that has an action "do: $e1$" and a leaving transition labeled by $e2$. If $e1$ is NULL, the do-action is missing. If $e2$ is NULL, the leaving transition is an automatic transition. Note that they both cannot be missing. If $e1$ or $e2$ is not missing, there is a corresponding leaving or arriving arc in a scenario. To simplify the presentation, a receiver of a sent event and a sender of a received event are not specified. It is assumed that the event name uniquely determines the receiver (the sender, respectively). This is true in most cases, but not necessarily always. For example, the same event can be received by (respectively, sent to) several object. In such cases, the name of the receiver (the sender, respectively) is simply appended to the name of the event (e.g., "e TO object1" or "e FROM object2").

Note that we essentially unify the concepts of an action and an event; from the receiver's point of view an arc in a scenario denotes an event, while from the sender's point of view an arc in a scenario denotes an action. Event arcs in scenarios may have guards associated to their names. Such guards can and should be associated to transitions but not to actions. Hence, Algorithm 1 cuts the guards off if the event is a sent one, but leaves them for received events.

The state name part s in the trace item triple $(e1, e2, s)$ is used in the synthesis algorithm given below. The algorithm gives the minimal state machine with respect to the number of states. However, this is not always what the designer wants. By giving state name boxes in scenarios the user may force the synthesis algorithm to separate states that would otherwise be merged. These state names can be seen in the resulting state diagram. If no state name boxes are given, no state in the resulting state diagram has a name.

We now proceed with a description of our state machine synthesis algorithm [54]. For that purpose we introduce selected definitions and terms. The concatenation of traces is defined in an obvious way. If

$$T_1 = (a_1, e_1, s_1) \cdots (a_m, e_m, s_m)$$

and

$$T_2 = (a'_1, e'_1, s'_1) \cdots (a'_n, e'_n, s'_n)$$

are traces, then T_1T_2 is a trace

$$(a_1, e_1, s_1) \cdots (a_m, e_m, s_m)(a'_1, e'_1, s'_1) \cdots (a'_n, e'_n, s'_n).$$

A state machine M is defined as a 5-tuple $M = (S, E, A, d, p)$, where S is a set of states, E is a set of events, A is a set of actions, *a transition relation* d is a partial function $d : S \times E \longrightarrow S$, and *an action relation* p is a partial function $p : S \longrightarrow A$. The transition relation defines the next state on the basis of the current state and the event. The transition defined by a partial function $d(s1, e) = s2$ is denoted as a triple $(s1, e, s2)$. The resulting state machine is deterministic, since the partial function d uniquely determines the next state. The action relation associates one (possibly empty) action with every state.

The algorithm given below is able to synthesize scenarios into a state machine incrementally (i.e., scenarios can be synthesized into an existing state diagram). If no editions have been made, the resulting state diagram is deterministic and minimal with respect to the number of states. However, the designer might have added new states and transitions that would not obey determinism. In order to respect designer's wishes, edited states and transitions are never removed during the synthesis, even when the algorithm needs to backtrack (in which case it might remove states that it just created). Every state/transition in a state diagram is either user edited or created by the synthesizer, the former being the default value. To distinguish the latter from the former, the synthesizer simply marks every state and transition it creates.

As an input, the algorithm gets a state machine $M = (S, E, A, d, p)$ and a trace $T = t_1 \dots t_m$. The trace items to be synthesized are read from a scenario. In addition to those trace items, the trace T contains all the (possible) trace items synthesized before. The trace is linked to the state diagram through actions of states; each action knows which trace items are currently linked to it.

In what follows, T is treated as a set of trace items. In the algorithm below, S' is a set of states, d' is a partial function $d' : S \times E \longrightarrow S$, p' is a partial function $p' : S \longrightarrow A$, q' is a partial function $q' : T \longrightarrow A$. The partial function q' links trace items to actions. Furthermore, $Used(t)$ gives the set of states so far considered for trace item t. A "free trace item" is a trace item that is associated with an action of an existing state but is not forced to be joined with it; the trace item could be associated with another action still not causing nondeterminism. The algorithm needs to backtrack if nondeterminism results or if there is no way to associate the trace items with states using the allowed number MAX of states. In the latter case, MAX is incremented. When backtracking, the synthesis information from the previously synthesized trace item down to the last free item is removed from the state diagram. After that another untried choice for synthesizing these items is taken. Note that Algorithm 2 presented below does not require an existing state diagram; the state machine given as a part of the input can also be empty.

Algorithm 2. Incremental state machine synthesis.

Input: A state machine $M = (S, E, A, d, p)$, trace $T = t_1 \ldots t_m$.

Output: Updated state machine $M' = (S', E', A', d', p')$.

Method:

```
Let $lfi$ be the position of the last free item in $T$.
Let $E' = E \cup \{e \mid (a, e, s) \in T\}$.
Let $A' = A \cup \{a \mid (a, e, s) \in T\}$.
Let $S' = S$.
MAX := $|S| + |A'| - |A| - 1$.
$i := 1$.
$untriedChoices := true$.
**while** $untriedChoices$ **do**
    **if** $lfi < 1$ **then** // does not exist
        Remove all synthesis information from $M'$.
        $lfi := t_1$.
        $i := 1$.
        MAX := MAX+1.
    **else**
        Remove synthesis information generated for trace items from $lfi$ to $i$ in $T$.
    **end**
    $i := lfi$.
    **if** there are free items in $T$ before $lfi$ **then**
        Let $lfi$ be the position of the last free item.
    **else**
```

```
  $lfi := 0$.
end
$Used(t_r) := \emptyset$, for $i < r \leq m$.
$validChoice := true$.
while $i \leq m$ and $validChoice$ do
// while the algorithm need not backtrack,
// i.e. no nondeterminism results
  $joined := false$.
  if $i > 1$ then
    $t_{i-1} := (a_{i-1}, e_{i-1}, s_{i-1})$.
    Let $st$ be the state for which $p'(st) = q'(t_{i-1})$.
    $s := d'(st, e_{i-1})$.
    if $s \neq NULL$ then
      if isJoinable($t_i, s$) then // forced join
        $q'(t_i) := p'(s)$.
        $i := i + 1$.
        $joined := true$.
      else
        $validChoice := false$. // backtrack
      end
    else
      $Cand$:=$\{s' \in S' \setminus Used(t_i) \mid$ isJoinable($t_i, s'$)$\}$.
      if $Cand \neq \emptyset$ then
      // free join, add new transition
        Pick up a member $s'$ of $Cand$.
        $Used(t_i) := Used(t_i) \cup \{s'\}$.
        $t_{i-1} := (a_{i-1}, e_{i-1}, s_{i-1})$.
        Let $st$ be the state for which $p'(st) = q'(t_{i-1})$.
        Add transition $d'(st, e_{i-1}) = s'$ to $M'$.
```

```
                Set the added transition to be synthesized.
                // synthesized transitions are removable during
                // the synthesis
                Set $t_i$ to be a free trace item.
                $i := i + 1$.
                $joined := true$.
            end
        end
    end
    if not $joined$ and $validChoice$ then
        if $|S| <$ MAX then
        // could not be joined, add a new state
            Create a new state $s'$.
            $S := S \cup \{s'\}$.
            Set $s'$ to be synthesized.
            // synthesized states are removable during
            // the synthesis
            $p'(s){:=}q'(t_i){:=}a_i$.
            $t_{i-1} := (a_{i-1}, e_{i-1}, s_{i-1})$.
            Let $st$ be the state for which $p'(st) = q'(t_{i-1})$.
            if $i > 0$ and $e_{i-1} \neq$ void then
                Add transition $d'(st, e_{i-1}) = s'$ to $M'$
            end
            $i := i + 1$.
        else
            $validChoice := false$.
        end
    end
end // while validChoice
```

```
    if $i > m$ then
        // all trace items have been synthesized
        $untriedChoices := false$.
    end
end // while untriedChoices
```

The following routine *isJoinable* is used by Algorithm 2. It simply decides if a trace item can be associated with a certain state and its action. Trace T and state machine M are readable from routine *isJoinable*.

isJoinable.

Input: A trace item $t_i = (a_i, e_i, s_i)$ and a state s of a state diagram.

Output:

true, if t_i can be joined with s,

false, otherwise.

Method:

```
Let $sn$ be the name of $s$.
Let $an$ be the name of the action of $s$.
if $s_i \neq sn$ or $a_i \neq qn$ then
    return false.
Let $st$ be the state for which $p'(st) = q'(t_{i-1})$.
if joining $t_i$ with the action $an$ would cause a loop then
// a transition $d'(st, e_i) = s$ (in which $e_i = NULL$) would cause
// a loop consisting of states and guardless automatic transitions
    return false. // this is needed only for free join
if $s$ has at least one leaving transition then
    if $e_i$ is guardless then
        if $s$ has a leaving transition with label $l$ such that exactly one of $e_i$ and $l$ is NULL
```

```
            then
                return false. // one of them is an automatic transition
        end
        if $e_i$ has a guard then
            if $s$ has a leaving transition with label $l$ such that $l$ has a guard similar to one in $e_i$ and
            exactly one of $e_i$ and $l$ consists of the guard only (an automatic transition with guard)
            then
                return false.
        end
    end
    return true.
```

Algorithm 2 implements the synthesis as a sequence of repeated attempts for associating trace items with states. If that cannot be done using the allowed number MAX of states, the process is repeated for MAX+1. The number of different actions is used as the initial approximation for the number of states. That is clearly a lower bound, since in the synthesized state diagram each state has at most one action.

In SCED the semantics of actions remain undefined, hence obstructing the synthesizer to treat all actions similarly. Automatic transitions, however, require more defined semantics for actions of states they are leaving from: an action of a state that has a leaving automatic transition has to run to completion without interruption of any event. In a synthesized state diagram, an automatic transition is always the only leaving transition of a state. This results when in a scenario there is a sent event (or an action box) immediately followed by another sent event (or an action box), from the point of view of the current object; the former event corresponds to an action of the source state of the automatic transition, and the latter one corresponds to an action of its target state. Since scenarios are sequential, these two events are indeed sent in succession. In forward engineering, if an action in a state with a leaving automatic transition is semantically a continuous one that stops only when interrupted, it can be seen as a design error in the scenario construction phase. Such situations do not occur in reverse engineering, because the event trace describes the actual run-time behavior of the system. It can even be argued that the synthesized state diagram is, in fact, emphasizing points in which the action needs to run to completion by themselves. The duration of the actions is insignificant.

5.3 Problems in the synthesis of state diagrams

The state machine synthesis has been treated as an inductive inference problem by Koskimies and Mäkinen [54]. Inductive inference studies algorithms for inferring rules from their application instances, usually considering functions and languages [3, 4]. Applying the BK-algorithm to the state diagram synthesis, however, differs from a typical inductive inference, causing inaccuracies to the synthesized state machine. For this reason, learning results of the BK-algorithm do not necessarily hold for state diagrams. The main source of problems is that scenarios, which all represent positive data (while examples in a typical inductive inference problem can be either positive or negative), do not give sufficient information about the state machine for inference. This often yields to too general state machines; in addition to example scenarios (positive data), the inferred state machine can execute several other scenarios as well. In other words, programs are more “complete” than state diagrams: there is usually a valid continuation for every possible combination of variable values in every point of a program (except after the halt statement), but there is usually not a valid transition for every possible event in every state of a state diagram. Overgeneralization is considered the most severe problem of inductive inference from positive data, since only negative data can expose too general guesses [4].

The fact that the synthesized state diagram may generalize the given scenarios is usually exactly the desired effect, but in some cases the result is not what the user expects. An example of the overgeneralization problem and means to solve it are studied next.

Consider a trace consisting of following trace items:

(show current time, set new alarm time, NULL), (show alarm time 5 secs, NULL, NULL),

(show current time, alarm time reached, NULL),

(buzzing,turn alarm off, NULL),

(show current time, set new alarm time, NULL),

(show alarm time 5 secs, NULL, NULL),

(show current time, VOID, NULL).

The minimal state diagram produced by using the pure BK-algorithm is shown in Figure 5.4. This

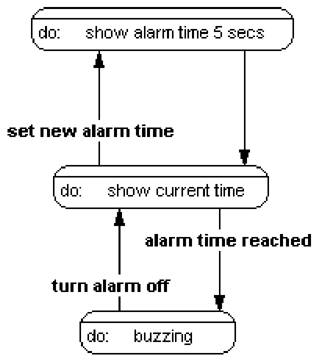


Figure 5.4: A minimal state diagram for a control unit of an alarm clock

is not a semantically acceptable solution, since the "*show current time*" phase is interrupted by the "*alarm time reached*" event even if the alarm is not set on. The algorithm has merged two logically different states together, thus overgeneralizing the state diagram. The state diagram also allows setting a new alarm time even if the previously set alarm time is not yet reached. That piece of information is not included in the event trace either. The latter example of overgeneralization describes the normal behavior of an alarm clock and is thus not harmful but the previous one is.

The overgeneralization problem can be solved in several ways. We could simply require that the scenarios should also cover forbidden transitions (negative data). That would equip the algorithm with sufficient information to avoid undesired state joins. However, this would be rather an inconvenient and unnatural design approach. Moreover, forbidden events never occur in reality, thus providing no solution for the problem from a reverse engineering point of view.

Using state boxes in scenarios is a slightly better attempt to avoid overgeneralization; by putting differently named state boxes before similar sent events (or similarly named action boxes) in scenarios, the user can make sure that the corresponding states will not be merged during the synthesis. In the previous example the trace items might look as follows:

(show current time, set new alarm time, ALARM OFF),
(show alarm time 5 secs, NULL, NULL),
(show current time, alarm time reached, ALARM ON),
(buzzing,turn alarm off, NULL),
(show current time, set new alarm time, ALARM OFF),
(show alarm time 5 secs, NULL, NULL),
(show current time, VOID, ALARM ON).

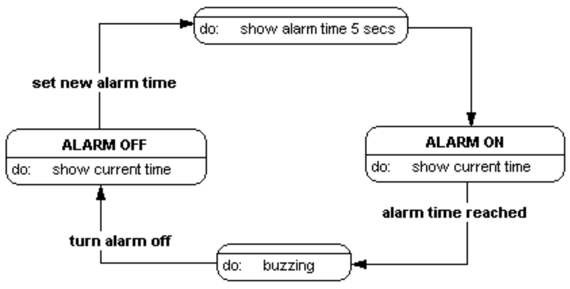


Figure 5.5: A state diagram with distinguishing state names synthesized for the control unit of an alarm clock

The synthesized state diagram with distinguishing state names is shown in Figure 5.5. Using state boxes is not always sufficient, and more importantly, it is usually not a convenient way to avoid the overgeneralization problem. Events corresponding to actions to be separated might appear in several scenarios. Even the small alarm clock example gives a hint of problems of this approach; since two states are joined only if they have similar state names, the designer might have to put state boxes in several scenarios to be absolutely sure that only right paths run through certain states. That can be very unpractical. State boxes are more useful, if the designer wants to emphasize that two (or few) paths should be separated from the main path. Unfortunately, this is rarely the case.

In reverse engineering, state boxes can be used to give additional information about the run-time behavior. For example, they can be used for naming branching points in the execution. In that case, the generated state boxes also speed up the state diagram synthesis by eliminating the number of possible merges and hence avoiding backtracking. The resulting state diagram also becomes more descriptive and understandable. Furthermore, such state boxes can be generated automatically and hence quickly. This unburdens the user from placing them manually in scenarios. Section 8.4.2 discusses the usage of state boxes in reverse engineering in more detail, and Chapter 9 gives some practical examples. It seems that state boxes are more useful in reverse engineering than in forward engineering.

Another approach to solve the overgeneralization problem is to use some heuristic rules to identify "suspicious" state merges, and ask the designer to either accept or reject the merge. This approach has its disadvantages as well. It is very difficult to find a satisfying rule for state separation, since the question is highly semantical. The rule might easily be too strong suggesting separation in too many cases; the number of suggested cases could be much larger than the number of harmful overgeneralization cases. If the rule is relaxed, all paths that need to be separated may not be found. Even if a fairly satisfying rule could be found, the designer has to choose between rejection and merging, perhaps several times, before the synthesis is completed.

The following "rule of flying visit" heuristic was implemented in SCED: a trace item was not allowed to be associated with an existing state if the trace both enters and leaves this state with new transitions. In this case the new trace makes only a flying visit to an existing path, reusing a single state. This can be assumed to be suspicious and less fruitful join [54]. However, it turned out that the "rule of flying visit" was often too strong; it suggested states to be separated in far too many cases. Furthermore, the number of the suggestions depends on the order in which scenarios are synthesized.

In SCED the only way to avoid a merge of two states before or during the synthesis is to use state boxes in scenarios. This approach has its disadvantages explained above, not to mention that trace items have to keep track of state names. However, state boxes appeared to be very useful in reverse engineering. Furthermore, the state diagram can be edited after the synthesis either manually or by using tools provided by SCED (cf. Section 4.2). For example, SCED offers tools for splitting and merging states. In addition, scenarios can be desynthesized out of the state diagram. Hence, detecting possibly overgeneralized states before or during the synthesis becomes less important. It also means that the responsibility of finding such states and separating them is left to the designer. This seems to be the most versatile approach to handle the overgeneralization problem.

5.4 The speed of the synthesis algorithm

The synthesis algorithm described in Sections 5.1 and 5.2 has an exponential time complexity because of the backtracking. Nonetheless, the algorithm is reasonably fast in practice because state diagrams are often rather small, and backtracking is not in heavy use. In fact, a state diagram can often be synthesized without backtracking at all. For instance, an ATM example (similar to one in [94]) consisting of 10 scenarios was synthesized in a fraction of a second, resulting in a state diagram with 15 states and 24 transitions. The example was run using a 120 Hz Pentium PC. However, synthesizing another artificially constructed example of approximately same size took a few tens of seconds to complete. The time needed does not directly relate to the size of the resulting state diagram, but rather to the state/action ratio; the larger the ratio is, the slower the algorithm tends to be. When the ratio is close to 1, the synthesis algorithm is fast.

Biermann *et al.* present some approaches for speeding up the synthesis algorithm: techniques for preprocessing the trace information to reduce enumeration, for pruning the search using a failure memory technique, and for utilizing multiple traces to the best advantage [6]. The algorithm that uses the failure memory technique keeps track of the cases in which merging states results in a failure and why these cases require backtracking. This allows the algorithm to avoid such failures later in the search. When applied for simple program synthesis tasks, the algorithm optimized with the pruning technique was even a bit slower than the original algorithm because the computational overhead for pruning cost more than it saved. For a more difficult synthesis problem, the saving in time was about 50% [6]. Some experiments using the failure memory technique were also made for the algorithm used in SCED. This approach indeed made the synthesis faster in unfavorable examples, but not dramatically; time consuming cases were too slow also with the modified algorithm. Furthermore, the modified algorithm requires more memory to be run because of the failure memory structure. It also might slow down the synthesis in simple cases because of extra computations. In addition, unfavorable cases are unusual in practice: a set of scenarios that describe nondeterministic behavior often refers to incomplete or erroneous design. In reverse engineering, such scenarios are rarely generated, especially, if state boxes are used to mark branching points in the execution (cf. Section 8.4.2). Because of the reasons above, there did not seem to be an urgent need for applying the speeding up techniques for the state diagram synthesis algorithm in SCED.

5.5 Limitations

A *timing mark* in UML is a denotation for the time at which an event or message occurs. Timing marks may be used in constraints [95] (p. 476). Timing marks and constraints in general are not included in SCED's scenario notation nor state diagram notation. However, they are used in several other related systems. For example, in a system developed for generating timed automata from timed scenarios, in which the requirements are given in a form of scenarios from which an automaton is synthesized [102, 103, 104]. Constraints can be seen as semantic conditions or restrictions that must hold at a certain point in the diagram. In SCED scenarios, such a constraint can currently be expressed by an assertion box. For taking different constraints properly into account in the synthesis, an interpretation mechanism would be needed. States and transitions depending on these constraints would be joined on the basis of the interpretation of the constraints. Similarly, interpretation mechanism for guards would be useful. For instance, a state that has two guarded leaving transitions *e[x < 15 sec]* and *e[x < 5 sec]* is not semantically reasonable, since the event triggers are the same and thus the former transition would never fire.

As mentioned earlier in this chapter, the synthesis algorithm can be applied to an operation code synthesis with the following assumptions: events in scenarios concern either operation calls or return events, and both the call events and the return events are shown. The latter assumption is needed, since otherwise there would be no way to define the end of an operation. Currently, a return event is defined for a call event the same way as a closing parenthesis is found for an expression consisting of nested parenthesized subexpressions. This is not a strictly acceptable way to define pairs of call and return events because the caller object could be called in between. If the SCED scenario notation contained control regions, as in UML sequence diagrams, the operation call sequences could be determined exactly. In Figure 5.6 an operation call sequence is shown using the notation presented in [95]. A double line shows the period of time when an object is active. This includes the entire life of an active object or an activation of a passive object, that is, a period during which an operation of the object is in execution, including the time during which the operation waits for the return of an operation that it called [95] (p. 424). A dashed line is used otherwise. Other UML sequence diagram constructs can, more or less, be expressed using the SCED scenario diagram notation. For example, recursion can be expressed by a repetition construct and conditional branching by a conditional construct.

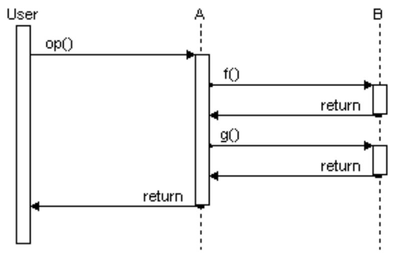


Figure 5.6: An event trace with control regions

5.6 Related research

In this section we discuss other approaches to transform example scenarios into a state machine. The basic notations related to the approaches are given but exact analyses of the algorithms are omitted.

Algorithms for synthesizing Real-time Object-Oriented Modeling (ROOM) models [100] are presented by Leue *et al.* [62, 63]. The MSC language in their approach contains basic MSCs (bMSCs) and high-level MSCs (HMSCs). A bMSC essentially consists of a set of processes (called instances in Z.120 [49]) that run in parallel and exchange messages in a one-to-one, asynchronous fashion. The processes in a bMSC correspond to participants in SCED. The graph of an HMSC provides operators for describing the composition and hierarchical arrangements of bMSCs. The synthesis generates the following structural ROOM components: *actors*, *protocols*, *ports*, and *bindings*. A resulting state machine is called a *ROOMchart*. In a bMSC, control passes from one local state of a process to the next one when an event is sent or received. When synthesizing a ROOMchart a local control state of a bMSC process is mapped to a state in a ROOMchart. As in our approach, synthesized ROOMcharts do not contain concurrency.

One of the questions addressed by Leue *et al.* is how many events should be executed during one ROOMchart transition [62, 63]. Two algorithms are developed corresponding to two possible answers: the transition terminates before the next received event (or whenever the last event in a bMSC) is encountered or the transition executes all sent events until the next received event is seen. The objective of the *Maximum Traceability Algorithm* is to preserve traceability of components of the synthesized ROOM behavior model with respect to the structure of the HMSC graph. The algorithm translates the HMSC graph into top level states of the ROOMchart associated with each actor. All edges in the HMSC graph are mapped onto transitions in the top level ROOMchart. A basic ROOMchart for each bMSC referred in an HMSC is created so that for every node in the HMSC graph, each incoming HMSC edge will be mapped onto exactly one incoming transition point in the corresponding lower level ROOMchart. In the *Maximum Progress Algorithm* events are sent during a single transition until the next received event is reached, which may possibly occur in a different bMSC. The maximum progress in every process will be different, and hence the synthesized state machines for different actors will not necessarily have identical structure (which is the case in the Maximum Traceability Algorithm). For each process, the algorithm first builds a directed graph for all sent and received events contained in all bMSCs. The message graph is then used to find a set of traces, which is then converted to a ROOMchart.

The main difference between our approach and the synthesis algorithm presented by Leue *et al.* is the usage of an HMSC [62, 63]. The graph of an HMSC connects bMSCs to describe their parallel, sequential, iterating, and non-deterministic execution. In our approach, scenarios represent example cases that can be treated in any order. Furthermore, the notation of a SCED scenario differs from the notation of a bMSC. Correspondingly, a SCED state diagram differs from a ROOMchart.

Somé *et al.* devise an incremental algorithm, used in implemented tools, for generating timed-automata from scenarios [103, 102]. The algorithm used for requirements engineering has been extended [102], for instance, for handling state explosion by grouping states. Their research combines scenario formalization and partial behavior composition. The scenario concept in their algorithm includes the initial situation of the scenario (*preconditions*), and a chain of interactions describing a system behavior as stimuli and reactions, associated with temporal constraints (timing constraints). The basic synthesis algorithm is built upon operation semantics: states are defined by *characteristic conditions* which hold in them and each operation execution results on *added-conditions* and *withdrawn-conditions*, the former being a set of conditions that become true, and the latter a set of conditions that are no longer true, after the execution of the operation. Although there are a lot of similarities in the approach presented by Somé *et al.* and in the one used in SCED, they also differ. The concepts of both scenarios and state machines are different. Furthermore, the timing constraints are not considered in SCED, but they are in a significant role in the system of Somé et al. Because of these differences, the logic and purpose of synthesis algorithms used differ as well.

Hsia *et al.* present a formalization of scenarios in which *scenario trees* describe a user view of a system [44]. A scenario tree consists of systems states (represented by nodes) and events (represented by edges between nodes). Hence, a scenario tree resembles a state diagram in Biermann's sense. It is required by Hsia *et al.* that each example path through a scenario tree (i.e., a scenario) starts from the initial state and ends up to a leaf node, and even more strictly, each scenario has to start and end in the same user-perceived system state. Such restrictions have not been made in SCED; the designer may define as long scenarios as she wishes, starting and ending by any event. Further, it is required in Hsia et al.'s system that there always has to be a labeled event between two states prohibiting the system to change state unless an event causes it. In UML and in SCED there can be an automatic transition between two states. On the basis of scenario trees, an abstract state machine can be constructed, which in turn can be used to verify inconsistencies, redundancies, and incompleteness in scenarios. As scenario trees, state diagrams have to start and end in the same state (i.e. there cannot be several final states). Again, in UML and SCED several final states may exist and none of them needs to be the initial state itself. In Hsia *et al.* 's system a state diagram can be used to generate other scenarios. Also in SCED implications to both directions are offered: a state diagram can be synthesized on the basis of information given in scenarios and a scenario can be traced from state diagrams [56, 111].

Glinz introduces a method for formal composition of scenarios into an integrated, consistent statechart-based model [38]. In this method every single scenario is modeled with a statechart [40]. These scenarios are then integrated into a single model, each scenario remaining visible as a building block in the model. Two scenarios, say *A* and *B*, can be related in four different ways: *B* after *A* (sequence), either *A* or *B* (alternative), *A* followed *n* times by itself (iteration), or *A* concurrent with *B* (concurrency). The scenario composition used in this method applies to disjoint scenarios only. Overlapping scenarios need to be either decomposed into mutually disjoint subscenarios or fused into a single scenario. The concurrency features included in this method can be helpful to accomplish the latter task. Because of different concepts of scenarios, the purpose of the method discussed by Glinz differs from the state diagram synthesis approach used in SCED.

An algorithm for synthesizing UML statechart diagrams from a set of collaboration diagrams is presented by Schönberger *et al.* [96]. It creates a statechart diagram for each distinct class implied by the objects in the collaboration diagrams. The algorithm maps both sent and received events of the object to transitions, while in our approach sent and received events have different roles in the synthesis. A statechart diagram is constructed by connecting a (ordered) set of transitions with states, split bars, and merge bars. Unlike our synthesis algorithm, the algorithm presented by Schönberg *et al.* is able to generate concurrent states. Another algorithm for integrating two statechart diagrams generated with the Schönberger *et al.* 's algorithm is discussed by Khriss *et al.* [53] and by Schönberger *et al.* [97]. This approach can be used to merge multiple statechart diagrams. As in our approach, the incrementality of the Schönberger et al.'s algorithm [97] enables an iterative process for dynamic modeling.

5.7 Summary

In SCED, the user can select one participating object from a scenario diagram and ask the generator to synthesize a state diagram automatically for it. The state diagram can be synthesized for one scenario only or for a specified set of scenarios. The synthesis algorithm is incremental, which allows the user to synthesize scenarios to an existing state diagram. Furthermore, the synthesis algorithm can be used when synthesizing operation descriptions from scenarios (instead of a state diagram).

The basic synthesis algorithm used has been presented by Biermann and Krishnaswamy [7], and its adoption to state machine synthesis from scenarios is discussed by Koskimies and Mäkinen [54]. This algorithm with a few modifications has been implemented in SCED.

Chapter 6

Optimizing synthesized state diagrams using UML notation

In this chapter we discuss information preserving means to transform a synthesized state diagram into a more compact form. We developed algorithms that detect similar responses to certain events and use that information to restructure the state diagram. The state diagram is modified by adding UML statechart notation elements into it.

Synthesized state diagrams are drawn in SCED using a rather primitive notation. No structuring has been done by means of UML statechart diagram concepts like special kinds of actions, generalization, and aggregation. The following definition characterizes a synthesized SCED state diagram.

Definition 6.0.1 *A plain state diagram is a minimal (with respect to the number of states), deterministic state diagram that consists of states and transitions between the states. Actions, which are all normal "do"-actions, are associated with states so that every state has at most one action.*

The UML statechart diagram notation is practical for specifying the dynamic model for a complex system. In addition to the capability to express concurrency and nested statechart diagrams, special kinds of actions can be used to make the diagrams more compact and readable. Such actions include entry actions, exit actions, actions fired by internal transitions, and actions attached to transitions. These concepts also emphasize similar responses to certain events and structure the statechart diagram according to them. For these reasons, it is desirable to add them to SCED state diagrams as well.

The process of adding the UML statechart notation (or at least part of it) to a plain state diagram is referred as *optimizing a plain state diagram* in the sequel. While optimizing a state diagram some states and transitions are removed. If a state with a (normal) action is removed, the action is moved and attached to another state or transition. This may happen only if the original state has an automatic transition as the only leaving transition (i.e., the action of the state must run to completion without interruption of any event). After the state diagram has been optimized, such actions might be used in a role in which they indeed need to be completed by themselves (e.g., as entry or exit actions). As in the case of the state diagram synthesis (cf. Section 5.2), the optimization algorithms do not assume anything about the duration of actions; the modifications concern only actions that have to be completed by themselves. Optimized state diagrams, perhaps even more clearly than synthesized plain state diagrams, point out the actions that cannot be interrupted.

In this chapter we show how and when internal actions, entry actions, and exit actions can be added to states, and actions can be attached to transitions. From now on, these actions will be called *UML actions*. We also show when actions from several states can be packed into a single state. The basic idea of combining information by packing actions and generating UML actions is to replace actions so that relations between actions and events causing them could be more easily seen. Combining information will also stress similar behavior of different paths through a state diagram. Figure 6.1 shows a plain state diagram synthesized from six scenarios. An optimized version of the state diagram is shown in Figure 6.2.

6.1 Definitions and rules

The following three rules are required to hold during the state diagram optimization process:

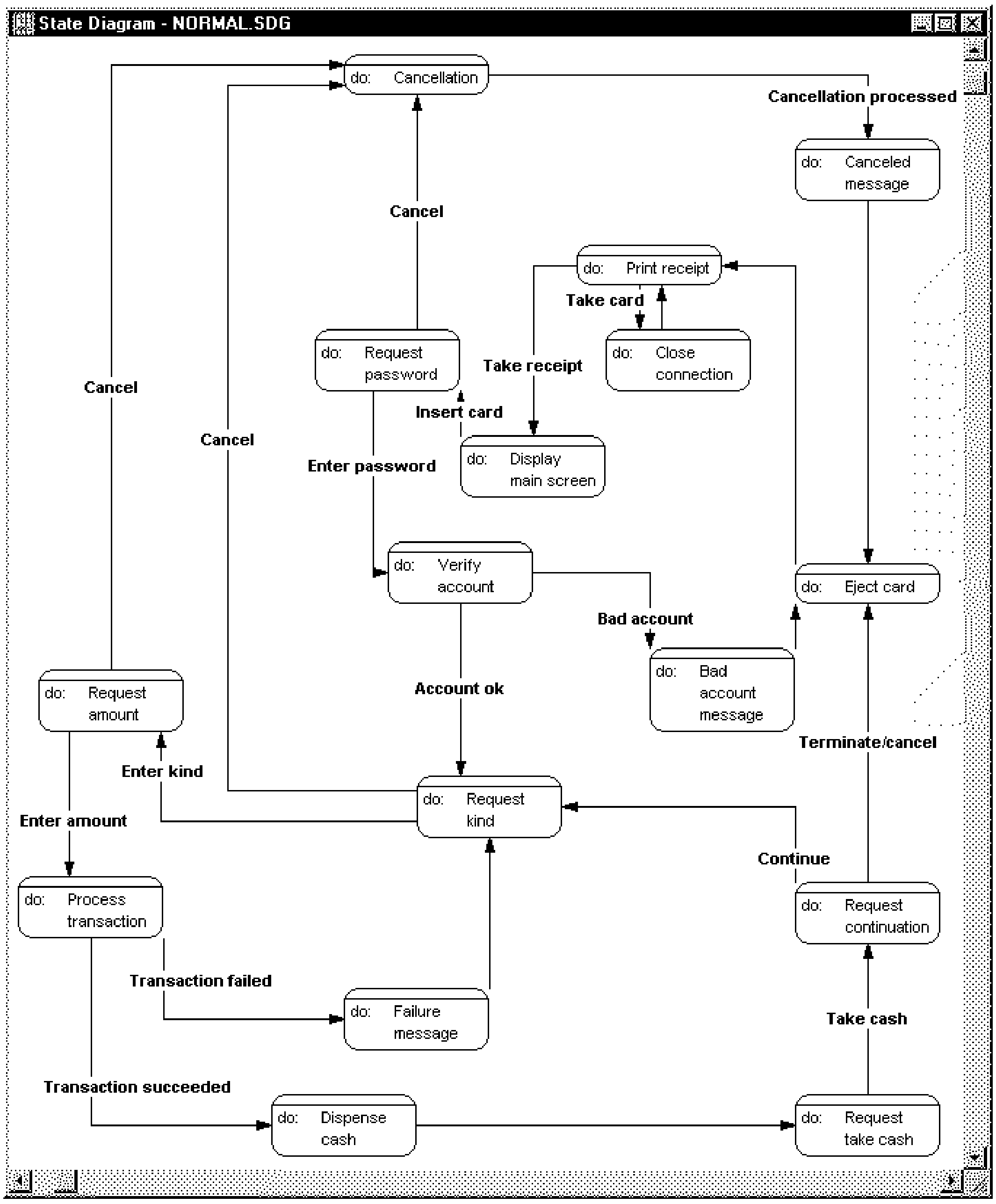


Figure 6.1: A synthesized plain state diagram

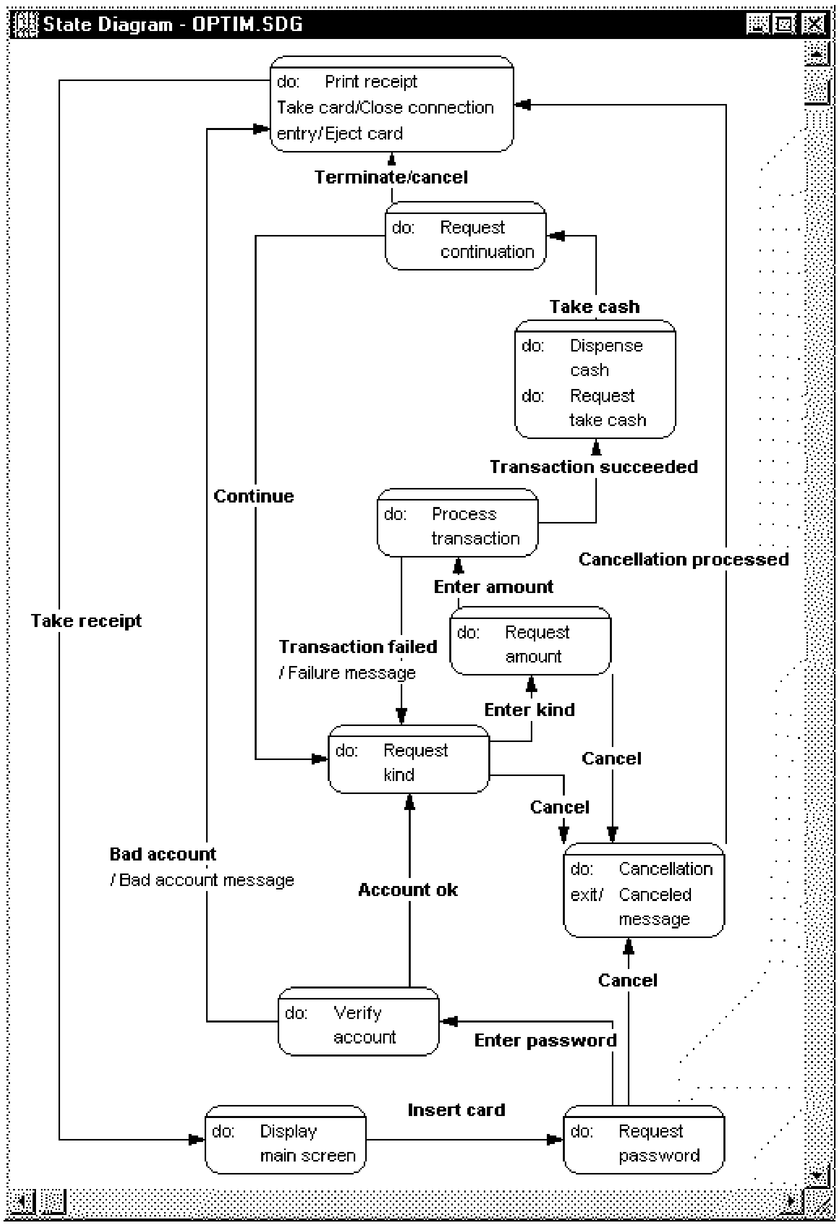


Figure 6.2: A state diagram optimized from the state diagram in Figure 6.1

Rule 6.1.1 *The information content of the state diagram should be preserved.*

Rule 6.1.2 *State diagram items should not be duplicated.*

Rule 6.1.3 *The number of states and transitions is minimized provided that Rules 6.1.1 and 6.1.2 hold.*

Rule 6.1.1 means that the optimized state diagram has to accept exactly the same scenarios as the plain one. It also means that the transformation of a plain state diagram, say G_p, to an optimized one, say G_o, has an inverse operation that changes G_o back to G_p. Rule 6.1.2 mainly concerns actions. While optimizing a state diagram some states and transitions are removed. For example, if a state with an action *a* is removed, then *a* should not be copied into several places. Instead, it should be attached to one state or one transition only. Rule 6.1.3 says that under the circumstances explained, UML statechart diagram concepts are added in such a way that the number of states and transitions will be minimized in the resulting optimized state diagram.

The theorems presented in this chapter define the optimization process. Optimization in SCED can be performed automatically by just choosing a specific command. However, different kinds of UML concepts can also be generated one by one, and the optimization can be applied to certain (selected) parts of the state diagram only. Hence, each theorem gives rules and requirements for two cases: the whole state diagram is supposed to be optimized, or only a certain concept is added (whenever possible). All Rules 6.1.1, 6.1.2, and 6.1.3 are required for the former case. The latter case is applied when the designer wants to use a certain subset of the UML state diagram notation (e.g., to generate exit actions). Then only Rule 6.1.1 needs to hold. Rules 6.1.2, and 6.1.3, instead, are not required; they just make sure that the resulting state diagram is minimal and still preserves the benefits of the state diagram synthesis.

For specifying characteristics of a UML state diagram, we now define abstract state diagrams. Their features apply to optimized state diagrams, as well as to plain state diagrams. An *abstract state diagram* is a 4-tuple, $OM = (S, E, A, d)$, where S is a set of states, E is a set of events, A is a set of actions, and d is a *transition relation* $d : S \times E \longrightarrow S$. Note that the formal specification of a state diagram presented in Section 5.2 (a 5-tuple $M = (S, E, A, d, p)$) cannot be applied to an optimized state diagram because of the action function $p : S \longrightarrow A$; states may have several actions in an optimized state diagram. For both variants of the same state diagram, the set of actions is exactly the same and the set of events may differ only by one event, namely the NULL event (corresponding to an automatic transition). Set S of an optimized state diagram may differ considerably from the one of a plain state diagram. Transition relation $d : S \times E \longrightarrow S$ is used in the same meaning for both kinds of state diagrams, since the next state can be explicitly defined by the current state and an event in both cases. A transition $d(s1, e) = s2$ can also be expressed in the form $(s1, e, s2)$.

The above definition of an abstract state diagram does not take all UML statechart diagram concepts into account. For example, it does not consider concurrency. However, the definition is sufficient for the discussion presented in this chapter.

Before defining the optimization process of a plain state diagram, some remarks are in order. These remarks are valid for both plain and optimized state diagrams, and will be applied later in this chapter.

The modified version of the BK-algorithm used in SCED (Algorithm 2 presented in Section 5.2) guarantees that in a plain state diagram both an unlabeled transition and a labeled one cannot be leaving transitions of the same state. The optimized variant of the state diagram should also have this property. This is formalized in Property 6.1.1.

Property 6.1.1 *Let $OM = (S, E, A, d)$ be an abstract state diagram. For each state $s \in S$, the following two conditions must hold:*

1. *If $d(s, NULL) = s1 \in S$ and there is an event e in E such that $d(s, e) = s2 \in S$, then $(s, NULL, s1) = (s, e, s2)$,*

2. *If there are events e and e' in E such that e is non-null, $d(s, e) = s1 \in S$, and $d(s, e') = s2 \in S$, then e' must be non-null, too.*

The definition of transition relation $d : S \times E \longrightarrow S$ itself ensures that the abstract state diagram is deterministic:

Property 6.1.2 *In an abstract state diagram, all transitions and their target states as well as all internal actions and their states are uniquely determined by an event and a state.*

The next property says that no loops consisting of states and automatic transitions can be found from an abstract state diagram. For a synthesized state diagram, this is guaranteed by Algorithm 2.

Property 6.1.3 *Let $OM = (S, E, A, d)$ be an abstract state diagram, and let $s_1, \ldots, s_n$ be states in S such that $d(s_{i-1}, NULL) = s_i, i = 2, \ldots, n$. If $d(s_j, NULL) = s', 1 \leq j \leq n$, then $s' \notin \{s_1, \ldots, s_j\}$.*

While adding special UML statechart diagram concepts to a plain state diagram, it is assumed that no state has a name. Such an assumption neither seems unrealistic, nor causes any major restrictions for the use of optimizing algorithms. First, names for the states are presumably given after the state diagram is in the desired form, since giving them in scenarios is not practical (cf. Section 5.3) and the synthesizer does not automatically generate them. Second, if there are state names (either given in scenarios or added manually to the plain state diagram), the same simplifying algorithms can be used with the restriction that states with actual state names may not be removed during the optimization process.

6.2 Packing actions

As mentioned in Section 4.1.2, it is assumed in SCED that all actions are executed in sequence and only the last action can be interrupted by a received event. The former condition is due to the fact that the state diagram is built using the sequential information given in scenarios. The latter condition results from the way actions are gathered into a single state: in the original plain state diagram such actions belonged to states that had automatic transitions as the only leaving transitions. Thus, all packed actions, except the last one, are considered to run to completion without interruption by any event. The latter condition can also be justified by the former one: if an event could interrupt the execution of actions at any point, then some actions (the last ones) might never be executed. Hence, the interpretation of such a sequence of actions is similar to the semantics of an action-expression in UML (cf. Section 3.4).

Due to the BK-algorithm [7], states are defined by their actions so that each state has at most one action. Gathering actions into a single state not only makes the state diagram semantically more reasonable but it also decreases the number of automatic transitions and the number of states. Such *action packing* is possible if there are automatic transitions between states that contain the actions, and the sequence of actions is approved by all paths in a state diagram. Theorem 6.2.1 gives requirements for packing actions of two states.

Theorem 6.2.1 *Let $s, s' \in S$ and $d(s, NULL) = s'$ in a plain state diagram. State s' may be removed and its actions can be moved to state s and placed after (normal) actions of s if and only if $(s, NULL, s')$ is the only transition entering s'.*

Proof 6.2.1 *According to Properties 6.1.1 and 6.1.2, transition $(s, NULL, s')$ is the only transition leaving state s. Packing is legal if $(s, NULL, s')$ is also the only transition entering s'. All paths through s continue immediately to state s', and s' cannot be entered in any other way. Finally, we have $s \neq s'$ according to Property 6.1.3.*

Conversely, suppose that there is another transition $(s_i, e, s'), e \in E, s_i \in S$ entering s'. Accord-

ing to Property 6.1.1, we have $s_i \neq s$. In order to satisfy Rule 6.1.1, action packing in this case requires that actions of the state s' are copied also to state s_i. This violates Rule 6.1.2. □

In terms of Theorem 6.2.1, all leaving transitions of state s' should be changed to leave state s after action packing.

Since several actions of a state are interpreted to be executed in succession, the results of a plain state diagram are valid for a packed one as well; such a sequence of actions could be seen as a single (more abstract) action, which is responsible for executing all the actions one by one (cf. an action-expression in UML in Section 3.4).

As an example of action packing, the state diagram in Figure 6.3 can be transformed into the state diagram in Figure 6.4.

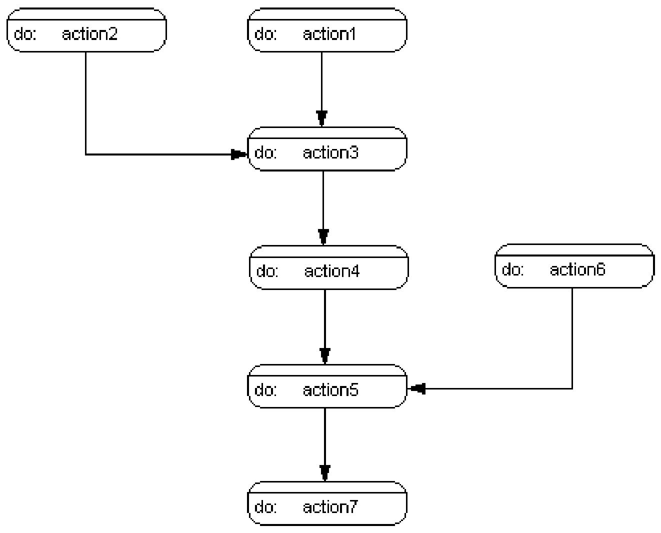


Figure 6.3: A state diagram in which each state has one action

6.3 Transformation patterns

Before presenting how UML actions can be added to a plain state diagram, we need to know what kind of state diagrams allow the transformations. For that purpose, two definitions clarifying the

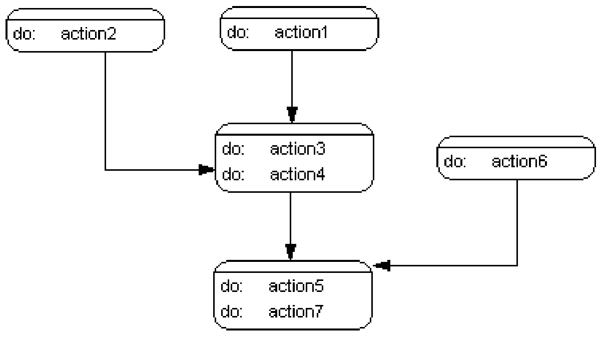


Figure 6.4: A state diagram resulting when applying the action packing operation for the state diagram in Figure 6.3

properties of plain state diagrams are introduced. These definitions are similar to the corresponding definitions for digraphs [32].

Definition 6.3.1 *A plain state diagram $G' = (S', E', A', d', p')$ is a* **subdiagram** *of a plain state diagram $G = (S, E, A, d, p)$ if and only if $S' \subseteq S$, $E' \subseteq E$, $A' \subseteq A$, $d'(s, e) = d(s, e)$ for all s in S' and e in E', and $p'(s) = p(s)$ for all s in S'.*

Definition 6.3.2 *Two plain state diagrams $G = (S, E, A, d, p)$ and $G' = (S', E', A', d', p')$ are* **structurally isomorphic** *if and only if $|S'| = |S|$ and there exists an order-preserving bijection $f : S \longrightarrow S'$ such that*

1. *If $e \neq NULL, e \in E, s_1, s_2 \in S, d(s_1, e) = s_2$, then there exists a non-null event e' in E' such that $d'(f(s_1), e') = f(s_2)$,*

2. *If $s_1, s_2 \in S, d(s_1, NULL) = s_2$, then transition $(f(s_1), NULL, f(s_2))$ belongs to G',*

3. *If $e' \neq NULL, e' \in E', s'_1, s'_2 \in S', d'(s'_1, e') = s'_2$, then there exists a non-null event e in E such that $d(f^{-1}(s'_1), e) = f^{-1}(s'_2)$,*

4. *If $s'_1, s'_2 \in S', d'(s'_1, NULL) = s'_2$, then transition $(f^{-1}(s'_1), NULL, f^{-1}(s'_2))$ belongs* *to* G.

Definition 6.3.2 says that two plain state diagrams are structurally isomorphic if they are structurally identical (i.e., only their existing labeling of transitions and actions of states may differ). That is, their adjacency matrices are equal up to a permutation of corresponding rows and columns.

Action packing removes automatic transitions and their target states. Such parts of a state diagram are removed also when modifying a state diagram by adding UML actions to it. Therefore, action packing and forming UML actions might be alternative ways to optimize a state diagram. In [110] it has been shown that the optimization operations can be used in such an order that Rule 6.1.3 will be satisfied.

Next, smallest subdiagrams (with respect to the number of states and transitions) that can be optimized by adding UML actions are introduced. The execution of any of the UML actions is (possibly) fired by an event. It is assumed in SCED that the execution of an internal action caused by a certain event is followed by the execution of normal actions of the state (cf. Section 4.1.2). The optional entry or exit actions are not executed. Hence, the execution of any of the UML actions is automatically followed by the execution of normal actions (if any) of a certain state. In the case of exit actions and actions attached to transitions, the optional entry actions of this state are executed right before its normal actions. Thus, adding any of the UML actions requires a plain state diagram in which there are states $s1$ and $s2$, a transition labeled with an event, say e, between them, i.e. $d(s1, e) = s2$, and an automatic transition leaving state $s2$ and entering a state, say $s3$, i.e., $(s2, NULL, s3)$. State $s1$ is the state before the execution of the UML action, event e of transition $(s1, e, s2)$ is needed to specify the event causing the execution of the UML action (especially in the case of an internal action), actions of $s2$ stand for the actual UML actions, and transition $(s2, NULL, s3)$ identifies the state the object moves to after the UML actions have been executed. Note that transition $(s2, NULL, s3)$ indeed cannot have any label, because otherwise $s2$ could not be removed without violating Rule 6.1.1. Now, states $s1$ and $s2$ cannot coincide because of Property 6.1.1, and states $s2$ and $s3$ have to be different states because of Property 6.1.3. If states $s1$ and $s3$ do not coincide, that is, their actions are different, then two possible subdiagrams exists. These subdiagrams are shown in Figure 6.5. From now on, the subdiagram on the left hand side is called *Subdiagram 1* and the one on the right hand side is called *Subdiagram 2*. Note that in Subdiagram 2, e could also be NULL (i.e. corresponding to an automatic transition).

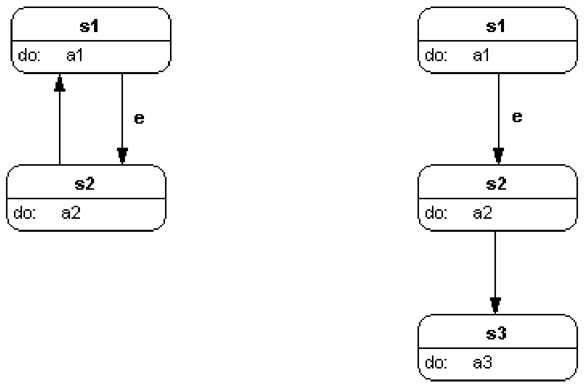


Figure 6.5: Subdiagram 1 (on the left) and Subdiagram 2 (on the right)

While adding UML actions it is assumed that no state has a state name. Yet, state labels $s1$, $s2$, and $s3$ are used in Subdiagram 1 and Subdiagram 2 in order to refer to a right state and to simplify the notation. Hence, $s1$, $s2$, and $s3$ are considered as identifiers rather than state names. Obviously, no state with an actual state name can be removed.

As mentioned in Section 6.2, the results for a plain state diagram are valid for a packed one as well, as long as we regard all normal actions of a state to be executed in a sequence without an interruption. It is worth noticing that after action packing Subdiagram 1 can have the form shown in Figure 6.5 only if there exists e' in E and s' in S such that $d(s', e') = s1, s' \neq s2$. Similarly, after action packing Subdiagram 2 can exist in the form shown in Figure 6.5 only if there exists e' in E and s' in S such that $d(s', e') = s3, s' \neq s2$.
Next we define the sets of neighbors for a certain state. The state itself is not allowed to belong to the set:

Definition 6.3.3 *In a subdiagram structurally isomorphic to either Subdiagram 1 or Subdiagram*

2, set $S_n(s)$ consists of states, to which there is a transition from state s, other than s.

A more general form of Definition 6.3.3 is the following:

Definition 6.3.4 *Set $S_{ng}(s)$ consists of states to which there is a non-looping transition from s, where s is a state in a plain state diagram.*

In what follows, we present several theorems concerning state diagrams having subdiagrams structurally isomorphic to Subdiagram 1 or Subdiagram 2. The following additional conditions are related to the theorems:

c_1 : $(s1, e, s2)$ is the only transition entering $s2$ (Subdiagram 1 or Subdiagram 2).

c_2 : $(s2, NULL, s1)$ is the only transition entering $s1$ (Subdiagram 1).

c_3 : $(s2, NULL, s3)$ is the only transition entering $s3$ (Subdiagram 2).

c_4 : In each $s' \in S_n(s1)$ there is an identical sequence of normal actions $a_1, \ldots, a_m, m \geq 1$, beginning from the first normal action a_1 of s' (Subdiagram 1 or Subdiagram 2).

c_5 : For each $s'' \in S_n(s1)$, $s1$ is the source state of each transition entering s'' (Subdiagram 1 or Subdiagram 2).

Further, we will later need the following conditions concerning any plain state diagram (condition c_6 was already used in Theorem 6.2.1):

c_6 : In a plain state diagram $G = (S, E, A, d, p)$, in which $d(s, NULL) = s'$, $(s, NULL, s')$ is the only transition entering s'.

c_7 : For a state $s \in S$ in a plain state diagram $G = (S, E, A, d, p)$ each $s'' \in S_{ng}(s)$ has an identical sequence of normal actions $a_1, \dots a_m, m \geq 1$, beginning from the first normal action a_1 of s''. Further, for at least one state s_i in $S_{ng}(s)$, either a_m is not the last normal action in s_i or s_i has an automatic transition as the only leaving transition.

c_8 : Let $s \in S$ be a state in a plain state diagram $G = (S, E, A, d, p)$. For each $s' \in S_{ng}(s)$, s is the source state of each transition entering s'.

In the following sections rules for forming UML actions are presented. These rules are divided into two cases: either only Rule 6.1.1 is required or both Rules 6.1.1 and 6.1.2 are required. Former cases give minimal conditions for forming UML actions. These conditions are applied when the user selects a state or a transition from a state diagram and "forces" the generator to form a desired UML action. However, forcing an UML action may restrict possibilities to finally get a minimal state diagram. Furthermore, it may cause copying information that was compounded by the synthesizer. Latter cases are used when the whole state diagram is optimized.

6.4 Internal actions

In SCED the interpretation of an internal transition *e/a2* of state $s1$ is the following: while being in state $s1$, if event e occurs, then actions $a2$ are executed and right after them all the normal actions of $s1$ but no entry nor exit actions (cf. Section 4.1.2). In UML, an activity has duration and can be interrupted. A transition that forces an exit from the controlling region aborts the activity. An activity is not terminated by the firing of an internal transition, because there is no change of state [95] (p. 133). In SCED, there is no specified way to distinguish actions and activities. However, the last normal action of $s1$ can be assumed to be interruptible by e, thus resembling the semantics of an activity. The interpretation of an internal transition in SCED hence imitates the original meaning of the internal transition in UML and thus provides a way to emphasize, at least in a very modest way, the different roles of entry actions and normal actions of a state. Furthermore, the chosen interpretation makes it possible to optimize a state diagram considerably as can be seen later in this section. It also stresses the difference between a (external) self transition and an internal action.

Figure 6.6 shows an example state diagram optimization using internal actions. Adding internal

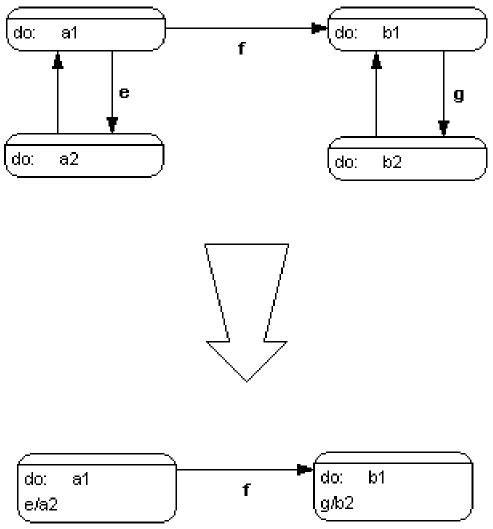


Figure 6.6: An optimization of a state diagram using an internal transition

transitions to a plain state diagram is possible only if it has a subdiagram that is structurally isomorphic to Subdiagram 1. Theorem 6.4.1 gives the exact condition. Note that the condition used in Theorem 6.4.1 was named c_1 previously in this chapter. Adding internal actions to Subdiagram 2 is obviously impossible, because the automatic transition leads to a state other than $s1$.

Theorem 6.4.1 *For a plain state diagram G, internal actions can be added to each subdiagram G', structurally isomorphic to Subdiagram 1, satisfying Rule 6.1.1 without restrictions, and satisfying both Rules 6.1.1 and 6.1.2 if and only if $(s1, e, s2)$ is the only transition entering $s2$.*

Proof 6.4.1 *According to Properties 6.1.1 and 6.1.2, an internal action can be formed regardless of any leaving transition of $s1$: $e' \in E, s' \in S$, and $d(s1, e') = s'$ imply that event e' is non-null and different from event e. No entering transition of state $s1$ (by the definition of an internal action) has any influence on forming an internal action. Because of Properties 6.1.1 and 6.1.2, the automatic transition has to be the only leaving transition of $s2$. Condition c_1 is obviously sufficient* *for forming an internal action for state* $s1$ *without violating Rules 6.1.1 and 6.1.2. Suppose then that* $e' \in E, e' \neq e$*, and* $d(s', e') = s2$ *for some* s' *in* S*. According to Properties 6.1.1 and 6.1.2,* $s' \neq s2$*. If* $s' = s1$*, then either of the internal actions e/a2 and e'/a2 or both of them can be formed satisfying Rule 6.1.1, but only by copying action(s) a2 and hence violating Rule 6.1.2. Suppose then that* $s' \neq s1$*. Then again internal action e/a2 can be formed without violating Rule 6.1.1, but only by copying actions a2 and hence violating Rule 6.1.2. In this case,* $s2$ *may never be removed without violating Rule 6.1.2.* $\square$

6.5 Entry actions

Figures 6.7 and 6.8 show two state diagrams that have been optimized by adding entry actions to them. They have subdiagrams that are structurally isomorphic to Subdiagram 1 and Subdiagram 2, respectively. When constructing conditions for adding entry actions, subdiagrams that are

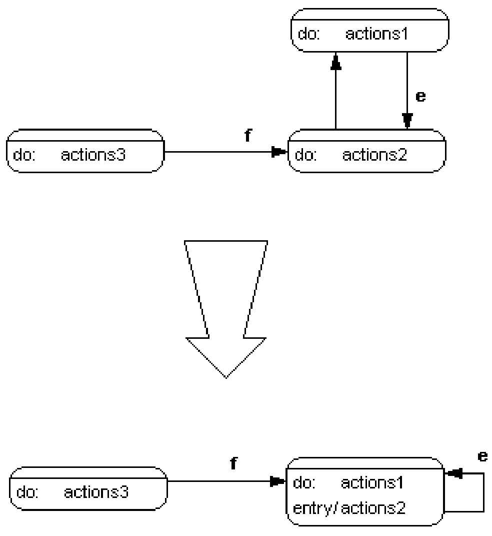


Figure 6.7: Using an entry action to optimize of a state diagram, which has a subdiagram that is structurally isomorphic to Subdiagram 1

structurally isomorphic to either Subdiagram 1 or Subdiagram 2 has to be found from a plain state diagram. Theorem 6.5.1 gives conditions for adding entry actions to a plain state diagram in the former case, and Theorem 6.5.2 gives corresponding conditions in the latter case. The conditions used in these theorems were named c_2 and c_3, respectively, previously in this chapter.

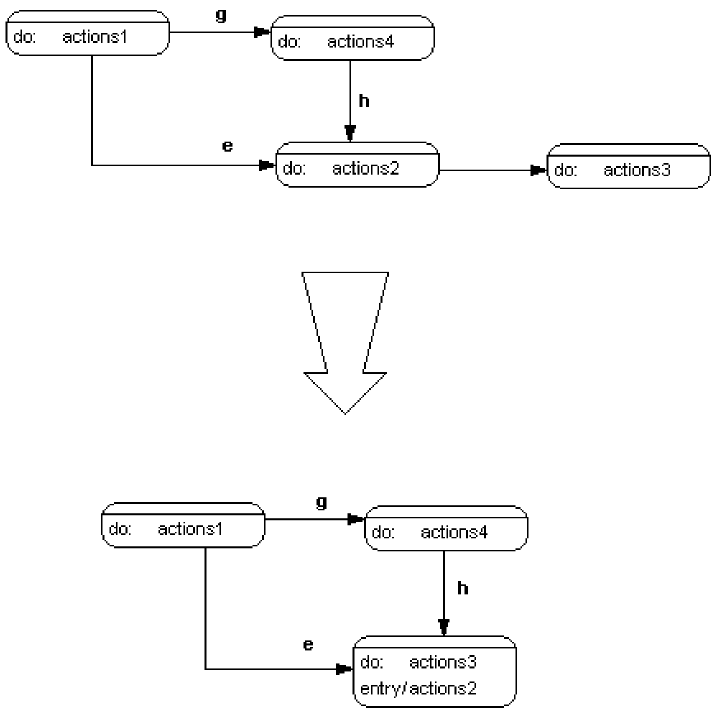


Figure 6.8: Using an entry action to optimize a state diagram, which has a subdiagram that is structurally isomorphic to Subdiagram 2

Theorem 6.5.1 *For a plain state diagram G, entry actions can be added to each subdiagram G', structurally isomorphic to Subdiagram 1, without violating Rule 6.1.1 if and only if $(s2, NULL, s1)$ is the only transition entering $s1$. In that case, also Rule 6.1.2 will be satisfied-*

Proof 6.5.1 *We first assume that there is a transition $(s', e', s1) \neq (s2, NULL, s1), e' \in E, s' \in S$. Transition $(s', e', s1)$ has to have a label ($e' \neq NULL$) or $s' \neq s2$, since otherwise the synthesis algorithm would have merged the source states s' and $s2$ (or at least the states corresponding last actions of s' and $s2$) and forced $(s', e', s1)$ to be $(s2, NULL, s1)$. Forming an entry action entry/a2 ($a2$ being actions of state $s2$) would not then be possible without violating Rule 6.1.1 because of transition $(s', e', s1)$.*

We next suppose that $(s2, NULL, s1)$ is the only transition entering $s1$. According to Properties 6.1.1 and 6.1.2, only entering transitions can be attached to $s2$ in addition to transitions of Subdiagram 1. Such transitions do not prevent us from forming an entry action satisfying both Rule 6.1.1 and Rule 6.1.2 as long as these transitions are changed to enter $s1$. Finally, leaving transitions of $s1$ do not influence on forming an entry action entry/a2 for state $s1$ (if they enter $s2$, they are changed to enter $s1$). $\square$

Theorem 6.5.2 *For a plain state diagram G, entry actions can be added to each subdiagram G', which is structurally isomorphic to Subdiagram 2, without violating Rule 6.1.1 if and only if $(s2, NULL, s3)$ is the only transition entering $s3$. Also Rule 6.1.2 will be satisfied if $(s2, NULL, s3)$ is the only transition entering $s3$.*

Proof 6.5.2 *As shown in Proof 6.5.1, an entry action entry/a2 cannot be formed for state $s3$ if $(s2, NULL, s3)$ is not the only transition entering $s3$. Thus, it is enough to show that the condition is sufficient. If condition c_3 holds, no leaving transition of state $s3$ or state $s1$ can enter $s3$. Hence, such transitions have no effect on forming an entry action for state $s3$. Transitions entering state $s1$ are also meaningless for forming the entry action. According to Properties 6.1.1 and 6.1.2, $s2$ can have only entering transitions attached to it in addition to transitions of Subdiagram 2. Such transitions do not prevent us from forming the entry action as long as we change these transitions to enter $s3$.* $\square$

6.6 Exit actions

Figure 6.9 shows how a state diagram can be optimized using an exit action. The original state diagram has subdiagrams that are structurally isomorphic to Subdiagram 1 and Subdiagram 2. Theorem 6.6.1 gives conditions for forming exit actions. Note that the conditions used in the

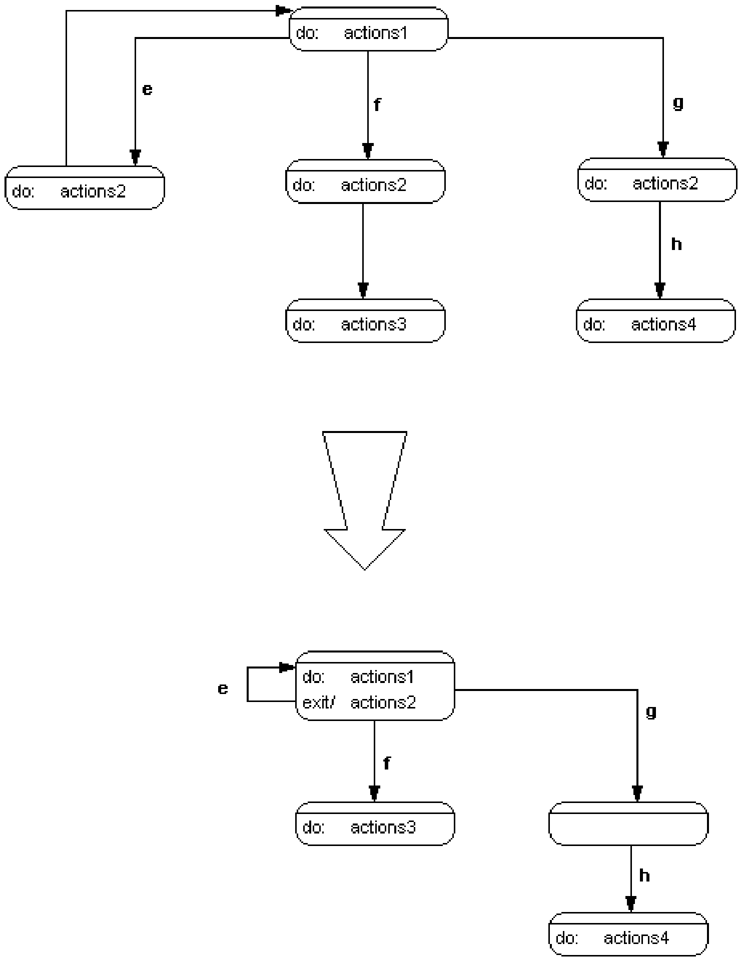


Figure 6.9: An optimization of a state diagram using an exit action

theorem were denoted by c_4 and c_5 earlier in this chapter. Definitions 6.3.3 and 6.3.4 are also needed.

Theorem 6.6.1 *For a plain state diagram G, exit actions can be added to each subdiagram G', structurally isomorphic to either Subdiagram 1 or Subdiagram 2, without violating Rule 6.1.1 if and only if c_4 holds. Exit actions can be added to G' satisfying Rules 6.1.1 and 6.1.2 if and only if* *both c_4 and c_5 hold.*

Let $a_1, \ldots, a_m$ be the longest sequence of normal actions as described in c_4. These actions are all moved to $s1$ as exit actions and removed from every $s'' \in S_n(s1)$. Consider a state $s_a \in S_n(s1)$ for which c_5 does not hold. In that case, a new state s_n is created, actions $a_1, \ldots a_m$ are copied into it, and every entering transition $(s_i, e_j, s_a), s_i \neq s1$ of s_a is changed to enter s_n, i.e. transition (s_i, e_j, s_n) is created. In addition, an automatic transition $(s_n, NULL, s_a)$ is added. Finally, if $s'' \in S_n(s1)$ has no actions left after $a_1, \ldots a_m$ have been moved to $s1$, and there is an automatic transition, say $(s'', NULL, s_m)$, then s'' and $(s'', NULL, s_m)$ are removed and all transitions entering s'' are changed to enter s_m.

Proof 6.6.1 *First we show that conditions c_4 and c_5 are needed. If c_4 is not true, there are two states, say s_f and s_g, in $S_n(s1)$ so that the first actions of their action sequences differ. It is then impossible to form an exit action for state $s1$ without violating Rule 6.1.1. Suppose then that there exist $s_f \in S_n(s1)$ and $s_p \in S$ so that $s_p \neq s1$ and there is a transition $(s_p, e'_p, s_f), e'_p \in E$ in the original state diagram. An exit action can then be formed without violating Rule 6.1.1 only by copying the actions of state s_f and either moving them to state s_p or to a new state that is created between states s_p and s_f. In both cases, Rule 6.1.2 would be violated. We still need to show that the condition $s1 \notin S_n(s1)$ of Definition 6.3.3 is required. If $s1 \in S_n(s1)$, then state $s1$ would also have actions $a_1, \ldots, a_m$, and there would be a self transition, say $(s1, f, s1)$. This means that the actions $a_1, \ldots, a_m$ are executed before event f is received. The Rule 6.1.1 requires then that if the exit actions are formed, then the actions $a_1, \ldots, a_m$ must also remain as normal actions in $s1$. In that case, the state diagram would not allow the following path, consisting of actions and events, any longer: $a_1, \ldots, a_m, f, a_1, \ldots, a_m, e, a_1, \ldots, a_m$.*

Next we will show that the conditions c_4 and c_5 are sufficient. No entering transition of state $s1$ has any effect on forming exit actions unless it is leaving state $s1$. Leaving transitions of states that do not belong to $S_n(s1)$ and which are entered from states that belong to $S_n(s1)$ by at least one transition (e.g., state $s3$ in the case of Subdiagram 2) have no influence on forming exit actions *for state $s1$. The same applies to the transitions that enter these states, except possibly the leaving transitions of state $s1$ or state $s2$. If c_5 holds, all entering transitions of states in $S_n(s1)$ leave state $s1$. Thus, leaving transitions of any $s' \in S_n(s1)$ cannot then enter any state in $S_n(s1)$ (which possibly will be removed). Hence, such transitions are meaningless when forming exit actions for state $s1$. If c_5 does not hold and there is a transition (s', e', s''), $s', s'' \in S_n(s1)$, then actions $a_1, \dots, a_m$ are executed twice when traversing along a path form $s1$ to s'' using (s', e', s''). That requires coping actions $a_1, \dots, a_m$ and hence violating Rule 6.1.2. Finally, condition c_4 guarantees that all leaving transitions of state $s1$ enter a state, actions of which can be moved to state $s1$. Other kind of leaving transitions of state $s1$ would make it impossible to form an exit action for state $s1$ without violating Rule 6.1.1.* $\square$

At the beginning of this chapter it was concluded that optimization algorithms replace only those normal actions that run to completion without interruption of any event; actions to be replaced are from states that have an automatic transition as the only leaving transition. Even though all states in $S_n(s1)$ are considered in Theorem 6.6.1, it is required that one of them belongs to a subdiagram that is structurally isomorphic to either Subdiagram 1 or Subdiagram 2. Let $s' \in S_n(s1)$ be such a state. Then the actions $a_1, \dots, a_m$ to be moved to $s1$ are always non-interruptible ones, since either there are still normal actions in s', which are automatically executed after a_m, or there is an automatic transition leading out of s', again causing some actions to be automatically executed after a_m. This implies that $a_1, \dots, a_m$ indeed need to run to completion without interruption of any event. Non-interruptible actions may appear also in other kinds of state diagrams. Figure 6.10, for example, shows a state diagram in which action $b1$ cannot be interrupted by any event and can hence be replaced by an exit action. This transformation combines information and hence clarifies the contents of the state diagram. However, it does not change the size of the state diagram. Hence, such transformations are recommended only if adding the exit action combines information, (i.e., there are at least two states whose actions are replaced by the exit action. The next theorem relaxes the rules for adding exit actions.

Theorem 6.6.2 *For a plain state diagram G, exit actions can be added to each subdiagram G' without violating Rule 6.1.1 if and only if c_7 holds. Exit actions can be added to G' satisfying both*

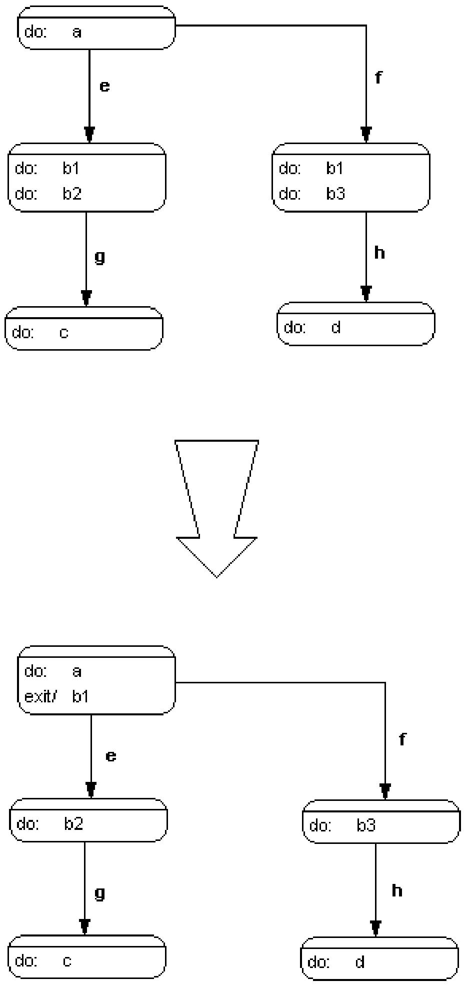


Figure 6.10: Modifying of a state diagram using an exit action

Rules 6.1.1 and 6.1.2 if and only if both c_7 and c_8 hold

The proof of the Theorem 6.6.2 is similar to Proof 6.6.1. Note that Theorem 6.6.2 still guarantees that actions $a_1, \ldots a_m, m \geq 1$, cannot be interrupted by any event, and can hence be moved to s as exit actions. It is also worth noting that rules for forming internal and entry actions cannot be loosened similarly; they can be added only for subdiagrams that are structurally isomorphic to either Subdiagram 1 or Subdiagram 2.

6.7 Action expressions of transitions

Figure 6.11 shows how a state diagram can be optimized by attaching an action to a transition. Theorem 6.7.1 gives the exact condition for adding action expressions for transitions. The con-

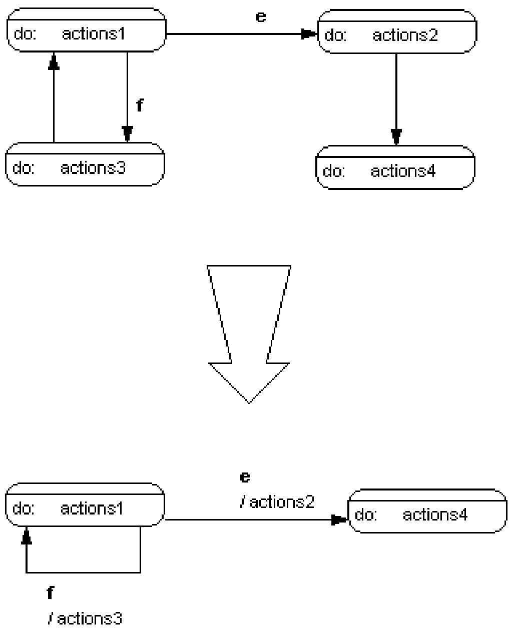


Figure 6.11: An optimization of a state diagram by attaching an action to a transition

dition used in Theorem 6.7.1 was also used in Theorem 6.4.1, and denoted by c_1 earlier in this chapter.

Theorem 6.7.1 *For a plain state diagram G, transition actions can be added to each subdiagram G', structurally isomorphic to either Subdiagram 1 or Subdiagram 2, without violating Rule 6.1.1. Transition actions can be added to G' satisfying both Rules 6.1.1 and 6.1.2 if and only if $(s1, e, s2)$ is the only transition entering $s2$.*

Proof 6.7.1 *We first consider transitions other than those in Subdiagram 1 or Subdiagram 2. Transitions attached to $s1$ but not to $s2$ have no effect on forming a transition action. The same applies to transitions attached to state $s3$ but not to state $s2$ in the case of Subdiagram 2. Further, according to Properties 6.1.1 and 6.1.2 the automatic transition has to be the only leaving transition of $s2$. Finally, transitions entering state $s2$ have to be considered. If $(s1, e, s2)$ is the only transition entering $s2$, then a transition action can be formed by removing $s2$ and the two transitions attached to it, and attaching the actions of state $s2$ to a new transition from state $s1$ to state $s1$ or to state $s3$, depending on whether Subdiagram 1 or Subdiagram 2 is considered. Assume then that there is a transition $(s', e', s2) \neq (s1, e, s2)$. Actions of state $s2$ can then be attached to transition $(s1, e, s2)$ satisfying Rule 6.1.1 regardless of transition $(s', e', s2)$. However, that is possible only by attaching actions of $s2$ to transition $(s', e', s2)$ as well, and hence violating Rule 6.1.2.* $\Box$

6.8 Removing UML notation concepts from state diagrams

When synthesizing new scenarios into an optimized state diagram, the state diagram is first *expanded* (i.e., transformed back to a plain one). This is necessary for the synthesis algorithm. In addition, the synthesis of a new scenario usually causes some new transitions and states to be created. These transitions and states naturally influence the optimization process. They might prevent some UML actions from being generated or enable a transformation that was not possible before. In practice, the new optimized state diagram includes most of the UML actions that existed in it before the synthesis. This is due to Property 6.1.1 and the fact that optimizing a plain state diagram is based on removing states that have automatic transitions as the only leaving transitions. For example, consider a state diagram that has been optimized by forming an internal action from a part structurally isomorphic to Subdiagram 1. It is highly probable that the same transformation is possible after synthesizing a new scenario, because the synthesis algorithm cannot add any other leaving transitions to state $s2$. However, it is possible that a new entering transition will be added to $s2$. That would prevent the optimizing algorithm to form the same internal action after the synthesis. This is possible only if the path described in a new scenario ends at $s2$. Hence, adding new scenarios into an optimized state diagram does not usually cause disturbing changes in the state diagram. In this section we discuss how UML state diagrams can be expanded.

As the optimization of a state diagram, the expansion can be produced using a single menu command in SCED. In the order presented above, each expansion method is applied to all possible subdiagram occurrences found in the state diagram.

After the state diagram has been expanded, the resulting state diagram is checked for any states that could be merged. Such states may occur if the generation of some UML actions have been forced (i.e., Rule 6.1.2 has been violated (cf. Section 6.1)). The steps to remove UML concepts from a state diagram are described next.

- Removing action expressions of transitions

  Suppose that in a state diagram there are states $s1$ and $s3$, and a transition $(s1, e/(a_1, \ldots, a_m), s3)$, $a_1, \ldots, a_m$ being actions attached to that transition. Then the actions $a_1, \ldots, a_m$ are detached from the transition by following the next steps:

  1. create a new state, say $s2$
  2. add actions $a_1, \ldots, a_m$ to $s2$ as normal actions
  3. add transitions $(s1, e, s2)$ and $(s2, NULL, s3)$
  4. remove transition $(s1, e/(a_1, \ldots, a_m), s3)$.

- Removing entry actions

  Suppose that in a state diagram there are states $s1$ and $s3$ and a transition $(s1, e, s3)$. Further, let $s3$ have entry actions $entry/a_1, \ldots, a_m$. Then the entry actions $entry/a_1, \ldots, a_m$ are removed from $s3$ using the following steps:

  1. create a new state, say $s2$

2. add actions $a_1, \ldots, a_m$ to $s2$ as normal actions
3. add transitions $(s1, e, s2)$ and $(s2, NULL, s3)$
4. remove entry actions $entry/a_1, \ldots, a_m$ from $s3$.

- Removing exit actions

  Suppose that in a state diagram there is a state $s1$. Further, let S_n be a set of states that are entered by at least one transition from $s1$. In addition, let $s1$ have exit actions $exit/a_1, \ldots, a_m$. Then the exit actions $exit/a_1, \ldots, a_m$ are removed from $s1$ by following the next steps for each $s' \in S_n$:

  1. if s' has no normal, entry, nor internal actions and all entering transitions of s' are leaving transitions of $s1$, then let sn be s', otherwise create a new state sn and create an automatic transition $(sn, NULL, s')$
  2. add actions $a_1, \ldots, a_m$ to sn as normal actions
  3. remove exit actions $exit/a_1, \ldots, a_m$ from s'
  4. for every transition from $s1$ to s', change the target state to be sn.

  In the first step sn is set to be s' if s' has no normal, entry, nor internal actions and all entering transitions of s' are leaving transitions of $s1$. This is done in order to avoid an empty state with an automatic transition as the only entering transition. Such a subdiagram can, of course, be modified also afterwards: all the leaving transitions of the empty state are changed to leave the source state of the automatic transition, and both the empty state and the automatic transition are removed. Note that such a subdiagram can never result after the synthesis. An empty state and an automatic transition may appear in a synthesized state diagram, but they cannot be consecutive. An empty state results if there are two received events in a row in a scenario diagram. Two successive sent events (or action boxes), on the other hand, cause a generation of an automatic transition.

- Removing internal transitions

  Suppose that in a state diagram there is a state $s1$ that has internal transitions

$$e/a_1, \ldots, a_m\ ,\ f/b_1, \ldots, b_n\ , \ldots,\ g/d_1, \ldots, d_p.$$

These internal actions are removed from $s1$ and replaced using the following steps for each individual internal action, say $e/a_1, \ldots, a_m$:

1. create a new state, say $s2$
2. add actions $a_1, \ldots, a_m$ to $s2$ as normal actions
3. add transitions $(s1, e, s2)$ and $(s2, NULL, s1)$
4. remove internal action $e/a_1, \ldots, a_m$ from $s1$.

- Unpacking actions

  Suppose that in a state diagram there is a state $s1$ with normal actions $a_1, \ldots, a_m$. Then the packed actions $a_1, \ldots, a_m$ are unpacked by following the next steps until there is only one action $a1$ in $s1$:

  1. create a new state sn
  2. move the last normal action from $s1$ to sn
  3. move all leaving transitions of $s1$ to leave sn
  4. add an automatic transition $(s1, NULL, sn)$.

Chapter 7

Rigi

Rigi is an interactive and visual reverse engineering environment [74]. It has been designed to help the user to understand and re-document software better. Rigi includes parsers to read the source code of the subject software. The extracted static information can be visualized and analyzed using Rigiedit, a graphical editor. Rigi runs on Sun SPARCstations (SunOS), IBM RISC System 6000 (AIX) workstations, and PC-compatible (Windows 95, Windows NT, Linux 2.x) machines.

7.1 Methodology

A semi-automatic reverse engineering approach of Rigi consists of two phases:

1. the identification of software artifacts and their relations
2. the extraction of design information and system abstractions.

The first phase is language dependent and is usually automated using parsers. Rigi provides parsers for C, C++, COBOL, and PL/AS, and the text formatting language LaTeX. The extracted information, consisting of software artifacts and their relations, is viewed as a directed graph with Rigiedit. The software artifacts are represented as nodes and their relations are visualized as arcs between the nodes. The types of nodes and arcs are defined in a language dependent domain model. For example, in this research Rigi has been customized for analyzing Java software. The extracted information includes the following Java software artifacts: classes, interfaces, methods, constructors, variables, static initialization blocks, and exceptions. These seven artifacts are represented as seven nodes types in Rigi (cf. Appendix A). The extracted dependencies between the artifacts 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. These relationships are represented as arcs between the artifacts (cf. Appendix B).

Rigi provides several layout algorithms that can be applied for the static dependency graph to view it in a desired way. The second phase is language independent and can be only partly automated. The system abstractions are constructed by generating hierarchical structures of subsystems [73]. Software quality criteria and metrics, such as the "low coupling and high cohesion" principle, are used for subsystem identification. Rigi supports program understanding also by providing algorithms and tools for generating different slices of the static dependency graph, that is, filtering out parts of the Rigi graph and thus providing a desired view of the target software system. This gives the user a possibility to focus on examining desired aspects of the software.

In Rigi, two key requirements, namely flexibility and scalability, have been addressed by making the system end-user programmable. A scripting language RCL has been incorporated into Rigiedit, allowing the user to automate and codify any step during the reverse engineering process. In addition, the user interface is fully customizable. The enhanced user control allows the user to tailor the environment to her own needs. Rigi can be used as:

1. a precursor for maintenance and re-engineering applications;
2. a front-end for conceptual modeling and design recovery tools;
3. an input to project decision making processes; and
4. a program understanding tool.

7.2 Rigi views

Rigi uses a directed graph to view the static information extracted from the source code of the target software: the software artifacts and their relations are represented as nodes and arcs in the static dependency graph, respectively. The type of a node or an arc is identified by a specific color. Various attribute values can be attached to both nodes and arcs. Node types, arc types, attribute types, and colors are defined in a domain model, written in four files Riginode, Rigiarc, Rigiattr, and Rigicolor, respectively (cf. Appendices A-C). The colorings for nodes and arcs can also be defined using the Rigiedit tool. The direction of an arc is not shown as an arrowhead, as is usually done in directed graphs. Instead, an arc from node A to node B is drawn as a line from the bottom of node A to the top of node B.

In Rigi, graphs can be nested to view the software on different levels of abstraction. The user can identify clusters of nodes (related according to some criteria) and collapse them into nodes that represent higher level components (e.g., subsystems). The new graph thus represents a more abstract view to the software, containing less nodes and less arcs. The nodes can be clustered to an arbitrary depth. By double-clicking a composite node, a new window pops up, showing the contents of that node, that is, the nodes and arcs that have been collapsed to it. The editor hence presents different levels in the hierarchy in separate windows. Because of that, the arcs connecting two nodes on different levels are cut off. Figure 7.1 shows an example of a hierarchical Rigi graph representing the structure of a C program. The most abstract view of the software (in this figure) is shown in the top window. It contains only two high-level nodes: RayTracer and ShaderLibrary. The arcs between these nodes are high-level *composite* arcs, which are derived from a bundle of one or more non-composite arcs. They indicate that the subsystem RayTracer is related to the subsystem ShaderLibrary, and vice versa. The content of the RayTracer node has been opened as a window on the left. Again, a composite node named Ray has been opened as the third window on the right. This window contains four method nodes (VoxelTrace, Trace, ShadeBackground, and CompteRayT) and three composite nodes (ObjectVoxel, VoxelSystem, and RayPrimitives) that could be opened further. The three dotted arcs are call arcs between the methods.

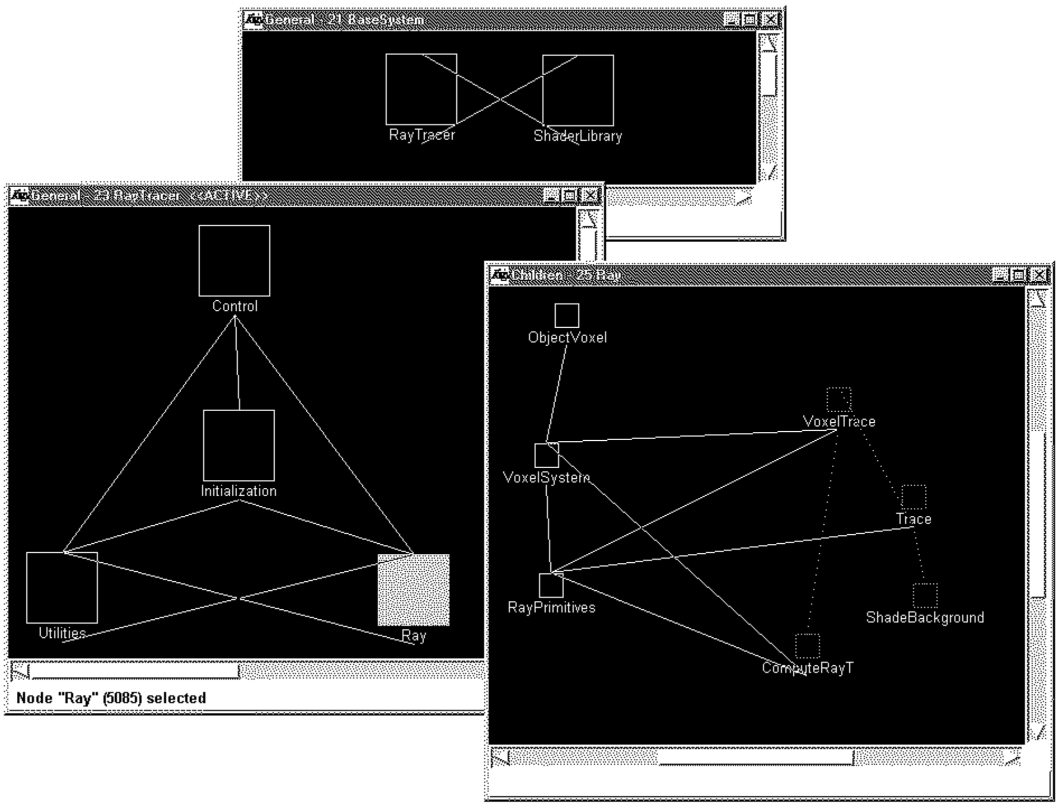


Figure 7.1: A hierarchical Rigi graph containing three levels of abstractions

To get an overview of the constructed hierarchies, a tree-like structure of the hierarchy can be generated and viewed in a separate window. Furthermore, a projection window can be generated to show the descendants of selected nodes.

Rigi provides also another approach for presenting hierarchical graphs, namely SHriMP (Simple Hierarchical Multi-Perspective) views [106, 107]. SHriMP views employ fisheye technique for nested graphs, showing the whole graph in a single view. Unlike Rigi views, a single SHriMP view is able to view the *level arcs* (i.e., arcs that are connecting two nodes on different hierarchical levels). In this research, the Rigi views have been used. There are three major reasons for that. First, information about the level arcs were not needed in a case study (cf. Chapter 9). Second, the static dependency graph resulting when extracting the information from the target Java software, FUJABA, was quite large (cf. Figure 9.1). A single SHriMP view might not be preferable over multiple Rigi views, if the software of that size is analyzed further [106]. Third, distinct parts of the FUJABA software were examined at one time in the case study. The information hiding facilities of Rigi were sufficient for constructing desirable static views in the example cases.

Rigi is able to view information stored in Rigi Standard Format (RSF). The RSF file format has two major dialects: unstructured and structured. In general, external tools, conceptual modelers, and parsers provide unstructured RSF for Rigiedit, and Rigiedit saves the graph as structured RSF, including spatial information such as the subsystem hierarchy [123].

An unstructured RSF file or stream consists of a sequence of triples, one triple on a line. Blank lines and comment lines starting with ♯ are allowed. The format for a triple is three optionally quoted strings: *verb source target* [123]. For example, a RSF file generated for a Java software could contain a triple *inherit de.uni_paderborn.fujaba.app.AboutBox javax.swing.JDialog*. The quotes are useful if the string contains white space characters.

7.3 Scripting

Rigi is extensible and end-user programmable [117]. The user can program Rigiedit and extend its facilities by writing Tcl/Tk [76] or Rigi Command Language (RCL) scripts. RCL is built on top of Tcl. Rigi provides an RCL script library that can be used to automate tasks, customize features, and integrate capabilities [123]. For example, the RCL library contains commands corresponding to each menu command in Rigiedit. The user can add new menu commands and thus customize the Rigiedit tool by writing new scripts. The RCL library also includes several scripts for manipulating the static dependency graph. Because Tcl is an interpretable scripting language, the script library of Rigi can be easily extended; new scripts can be written and added to it, on-line if desired. This allows the user to write and use scripts that have specific tasks.

In this research, Rigi has been customized to analyze Java software. The RCL library has been extended in various ways to support this task. Scripts for the following purposes have been added to the RCL library:

1. reading files that follow an extended RSF format, in which attribute values for arcs are allowed (cf. Section 8.6),
2. evaluating the graph using object-oriented software metrics (cf. Section 8.3),
3. normalizing the metrics values [113],
4. building high-level graphs by taking advantage of Java language structures (cf. Section 8.9),
5. slicing the graph by SCED scenario diagrams (cf. Section 8.8), and
6. writing information about the graph into files (needed by JDebugger, see Section 8.4).

Most of the scripts written are targeted for analyzing and modifying a graph that represents a structure of a Java software system. Furthermore, some scripts are useful only if the graph views the software artifacts and the relations extracted from the byte code by JExtractor (cf. Section 8.2), that is, the graph follows the domain model described in the Appendices A-C.

7.4 Reverse engineering object-oriented software using Rigi

UML class diagrams are widely used for modeling the static structure of object-oriented software systems in both forward and reverse engineering. In this research, Rigi has been used for static reverse engineering. The extracted static information of Java software is viewed as directed graphs. The static dependency graph contains approximately the same information as a class diagram. In Rigi, classes and interfaces have their own node types. Methods, constructors, static initialization blocks, and variables given inside a class in the UML class diagram are shown as nodes that are connected with a *contains* arc from the class node in Rigi. Each node that represents such a class member has a return type in a Rigi graph. For a method it is the type of the returned value, and for a variable it indicates its type. The generalization relationships are shown as *inherit* arcs in Rigi. Associations are shown in more detail in Rigi. While the class diagram shows an association between two classes, Rigi shows the individual relationships that cause two classes to be associated. Such relations include *call* relationships between member functions (i.e, methods, constructors, or static initialization blocks) and *access* or *assignment* relationships between a method and a variable. Information about multiplicity of the association can be extracted from those relationships in Rigi. A composition between two classes can not be directly expressed in Rigi. If the composition is implemented as variables (i.e., instances of the aggregated components are expressed as variables in the owner class), the *contains* relationships and the types of the variables in Rigi can be used to conclude information about the composition. The following Java keywords can be shown in both of the models: public, protected, private, final, static, abstract, native, final, volatile, and synchronized. Table 7.1 enumerates the main UML class diagram constructs and the constructs that can be used in Rigi for expressing the meaning of the UML class diagram construct in question.

In this research, we had several reasons for choosing Rigi. First, Rigi scales up. When reverse engineering the FUJABA target system, which contains about 700 classes, the static analysis produced 25854 software artifacts (cf. Figure 9.1). Viewing that information in a form of a single class diagram would be difficult.

A UML class diagram construct	A Rigi static dependency graph construct	Correspondence
class	Class (node type)	replacing
interface	Interface (node type)	replacing
method	Method (node type)	replacing
constructor	Constructor (node type)	replacing
static initialization block	Staticblock (node type)	replacing
variable	Variable (node type)	replacing
exception (stereotype)	Exception (node type)	replacing
generalization	Inherit (arc type) and	replacing
	Implement (arc type)	
composition		replacing
	call (arc type)	partly missing from UML
	access (arc type)	missing from UML
	assign (arc type)	missing from UML
send (stereotype)	throw (arc type)	replacing
class compartments	contains (arc type)	replacing

Table 7.1: UML class diagram constructs and the constructs that can be used in Rigi for the task in question. The correspondence is characterized as "replacing" if such a Rigi construct exists and "missing" otherwise.

Second, Rigi allows, for example, viewing variable accesses and method calls, which is not possible using standard class diagrams. Hence, information that can not be expressed using a class diagram can be included in a Rigi view. In this research, such pieces of information are used for guiding the dynamic reverse engineering process. In addition, including information about the member function calls enables merging dynamic code usage information into the view. The code coverage information, in turn, can be used for analyzing the software in various ways.

Third, Rigi supports building high-level views of the software by constructing hierarchical structures of Rigi views. Such abstractions are harder to construct for class diagrams because the notation for class diagrams is richer than the one for directed graphs. Dósa and Koskimies [30] introduce an approach for extracting high-level views of UML class diagrams. The used technique compresses information from the class diagram. An incremental expanding mechanism can be used to get back the original class diagram.

Fourth, Rigi provides a large and extensible set of slicing mechanisms. They help the user to focus on a desired aspect of the software. In this research, the slicing algorithms are used, for instance, to define the software artifacts for which dynamic information is generated.

Fifth, Rigi is easy to customize and extend, hence providing a good environment for the experiment described in this dissertation. Reverse engineering is a complex process. Usually, not only one tool but a set of tools is used for this purpose. Hence, for flexible integration, it is highly desirable that the software exploration tools are programmable and customizable and can thus be adopted and plugged into a new environment.

7.5 Summary

Rigi is an interactive and end-user customizable reverse engineering environment. Rigi includes parsers to read the source code of the subject software. The extracted information is visualized as a directed graph using Rigiedit. The software artifacts and their relations are visualized as nodes and arcs, respectively. The type of a node (e.g., a method or a variable) and the type of an arc (e.g., a method call or a variable access) are represented by specific colors. Rigi is able to view information stored in Rigi Standard Format (RSF) [123].

A scripting language RCL [123], which is built on top of Tcl [76], is incorporated into the Rigiedit tool. Rigi provides an RCL script library that allows the user to analyze and modify the static dependency graph. The RCL library can be extended by new Tcl/Tk or RCL scripts, on-line if desired. The scripts can be used, for instance, to build high-level views of the software. The user can identify clusters of nodes and collapse them into nodes that represent higher level components (e.g., subsystems). The new graph thus represents a more abstract view of the software, containing less nodes and less arcs. The nodes can be clustered to an arbitrary depth. The contents of the collapsed nodes can be opened in separate windows. The scripts can also be used to filter out information from the graph, to make queries on the graph, to read information from a RSF file, to write information to a RSF file, etc.

In this research, Rigi is customized for analyzing Java software. The information is extracted from Java byte code files and stored in RSF files. The RCL library has been extended in various ways to support static reverse engineering of Java software systems.

Chapter 8

Applying Shimba for reverse engineering Java software

Shimba is an environment for reverse engineering Java applications and applets. It is written in Java and uses the jdk 1.2 Virtual Machine (JVM). It can be used to collect and to view static and dynamic information from the software. Furthermore, a metrics program can be run to calculate selected object-oriented metrics values from the collected information. The metrics values can be used for analyzing the extracted information further.

8.1 Overview of the implementation

In Shimba, Rigi is used for viewing and analyzing the static structure of the software extracted from the Java byte codes comprising the subject system. SCED is used for modeling the event trace and dynamic control flow information generated when running the target system under a customized jdk debugger. To generate dynamic control flow information, breakpoints are set for the debugger on the lines that indicate branching caused by conditional statements. The line numbers for this purpose are extracted from the byte code. Dynamic code coverage information can also be attached to the static Rigi view when debugging a target software system. Information between the constructed views can then be exchanged. The constructed views can then be used to modify and improve each other by means of information exchange, slicing, and building abstractions, as discussed in Sections 8.7, 8.8, and 8.9. Figure 8.1 shows the overall structure of Shimba.

Traditionally, the static information is parsed from the source code. We had three main reasons for choosing Java byte code instead. First, extracting the static information from the byte code allows Shimba to be independent of the source code. 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 quick since no parsing needs to be done.

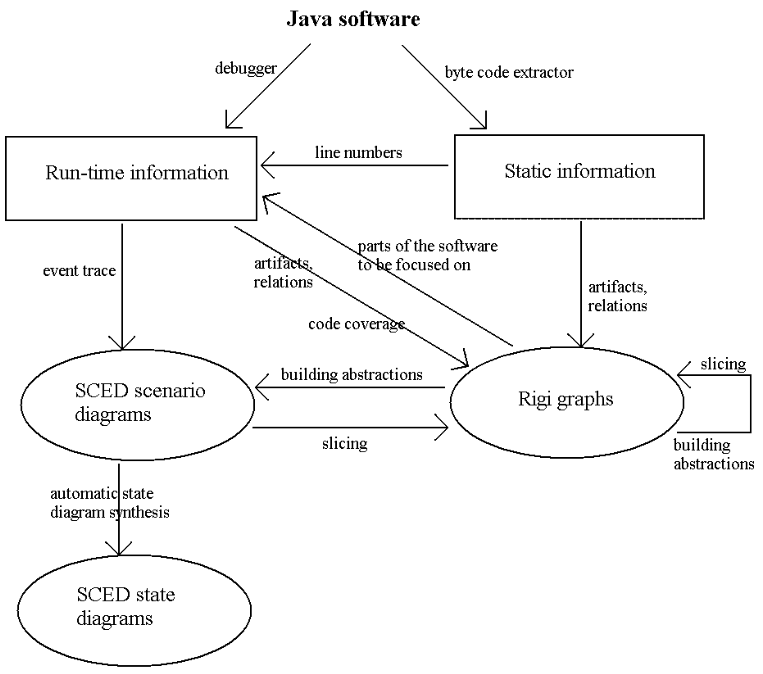


Figure 8.1: Shimba — An environment for reverse engineering Java software

8.2 Constructing a static dependency graph

Static software artifacts and their relationships are read from Java class files. The implementation of the extractor, called *JExtractor*, was written in Java by the author. It uses some of the public classes of the *sun.tools.java* package of jdk1.2. The static information is extracted following the descriptions of the contents of Java class files [118]. The extracted information includes the following Java software artifacts:

1. classes,
2. interfaces,
3. methods,
4. constructors,
5. variables, and
6. static initialization blocks.

Some attribute values are attached to the above software artifacts during the static analysis of the software. These attributes and their possible values are listed in Table 8.1. Some metrics values can also be added as attribute values to the software artifacts. This is discussed in Section 8.3 in more detail.

An attribute	Added for	Values
visibility	classes, interfaces member functions	public, protected, or private
return type	member functions	a string
package	classes and interfaces	a string
static	member functions	1 or 0
abstract	member functions	1 or 0
native	member functions	1 or 0
final	member functions	1 or 0
volatile	member functions	1 or 0
synchronized	member functions	1 or 0

Table 8.1: Software artifacts and their attribute values extracted during the static analysis of the software. The phrase “member functions” refers to methods, constructors, and static initialization blocks.

The extracted relationships among the software artifacts are:

1. an extension relationship (e.g., a class extends another class),
2. an implementation relationship (a class implements an interface),
3. a containment relationship (e.g., a class contains a method),
4. a call relationship (e.g., a method calls another method),
5. an access relationship (e.g., a method accesses a variable), and
6. an assignment relationship (e.g., a method assigns a value for a variable).

No attribute values are attached to the relationships in the static analysis of the software. However, call relationships, for example, can be given weight values based on the run-time usage of the software. Furthermore, during the dynamic analysis of the software exceptions and throw relationships (e.g., a method throws an exception) can be added to the static dependency graph. Adding dynamic information to the static dependency graph is discussed in Section 8.6 in more detail.

The user can ask the JExtractor to read information for:

1. all classes in a specified directory (optionally all the subdirectories are included),
2. selected classes (a list of classes is given as a file),
3. all classes in a single class file (a class file usually includes one class definition only), or
4. all classes in a single class file and all their dependencies (i.e., all classes that are used by any of the previously parsed classes).

The extracted static information is currently stored in Rigi Standard Format (RSF) [123]. Shimba uses an internal graph representation, which temporarily stores the extracted information before dumping it into a file. All static information and the part of dynamic information that is added to the static dependency graph is stored in this graph. A specific writer is implemented to write the contents of the graph to a file in RSF format. Hence, the saving format is not hard-wired in the implementation of JExtractor. Another writer can easily be implemented to support another saving format.

The resulting RSF file can be loaded into the Rigi reverse engineering environment using a single menu command. In Rigi all the software artifacts are shown as nodes, and relationships as directed edges between two nodes.

8.3 Software metrics used in Shimba

Software metrics can play a significant role when reverse engineering an existing software system. One approach is to use object-oriented complexity metrics to identify high- and low-complexity parts of the subject system. The most experienced maintainers or reengineers can then be assigned to the most complex subsystems [116]. Another strategy is to identify complex or tightly coupled parts in the subject software system. Such parts are difficult to modify and reuse and might be candidates for restructuring, refactoring, or significant redesign [35]. Metrics can also be used to identify highly cohesive and loosely coupled parts of the software that potentially represent subsystems [73]. Measures of the inheritance hierarchy of an object-oriented software system can give good predictions of reusability and design complexity.

A metrics program has been integrated with Shimba [113, 114]. The user can select any combination of the seven metrics listed below and give them to the metrics program as parameters. The metrics program calculates values for the selected metrics from the information extracted by the byte code parser. The metrics suite contains seven product metrics that can be used to estimate and measure inheritance relationships, complexity, and communication of the target Java software [113, 114]. The metrics provided are:

1. Depth of Inheritance Tree (DIT),
2. Number of Children (NOC),

3. Response For a Class (RFC),
4. Coupling Between Objects (CBO),
5. Lack of Cohesion in Methods (LCOM),
6. Cyclomatic Complexity (CC), and
7. Weighted Methods per Class (WMC).

For each metric, Table 8.2 enumerates the software artifacts it is applied to, its primary focus, and aspects it measures. The calculation of the metrics is described in the Appendix D.

Metric	Calculated for	Primary focus	Things to measure
DIT	classes, interfaces	Inheritance	reuse, understandability, and testability
NOC	classes, interfaces	Inheritance	reusability and potential influence on the design
RFC	classes	Communication	coupling, complexity, and requirements for testing
CBO	classes	Communication	coupling, reusability
LCOM	classes	Communication	cohesion, complexity, encapsulation, and usage of variables
CC	classes, member functions	Complexity	complexity and branching in the control flow
WMC	classes	Complexity	complexity, size, effort for maintenance, usability, and reusability

Table 8.2: Seven software product metrics, software artifacts they are applied to, their primary focus, and things they measure.

The metric values calculated can be dumped into a file or added to a Rigi graph as node attributes and used for analyzing the software. In Rigi Tcl/Tk [76] scripts can be run on the static dependency graph [123]. The scripts can be used to make queries about the graph or to modify it. The scripts provide a flexible way to analyze the metric values. For example, by running a script the user can easily focus on parts of the software that have metric values in a desired value range. Because Tcl is an interpretable scripting language, the script library of Rigi can be easily extended; new scripts can be added to it, on-line if desired. This allows the user to write and use scripts that have specific tasks (e.g., scripts that support the analysis of the metric values).

8.4 Collecting dynamic information

When reverse engineering Java software using Shimba, the dynamic event trace information is generated automatically as a result of running the target system under a customized jdk debugger, called *JDebugger*. The implementation of JDebugger uses public classes of the *sun.tools.debug* package of jdk1.2. It generates event trace information about method calls, constructor invocations, and thrown exceptions. Such events can be sent and received by methods, constructors, or static initialization blocks of classes. In the sequel, these class members are called *member functions* in short. In addition, information about the dynamic control flow, that is, a dynamic slice of the implicit (static) control flow, can be generated if desired. The dynamic control flow describes branching within member functions. It consists of a notification of an executed conditional structure and an acknowledgement of the result. In Java such conditional structures are **if**, **for**, **while**, and **do-while** statements. The branching information caused by a **switch-case** statement is recorded as well. In the event trace generated the receivers and senders of events, as well as owners of conditional structures, can be set to be either classes or objects. When loading the event trace information into SCED, they are represented as scenario participants and shown as vertical lines. In some cases it is enough to examine the object interaction classwise, while in other cases it is necessary to distinguish among instances of classes (i.e., study the behavior at the object level). In the former case, the scenario participants are classes, in the latter case they are objects.

8.4.1 The event trace

To capture the event trace information, the user can ask JDebugger to set breakpoints at the first line of

1. a set of selected member functions (the set is given as a file),

2. all member functions of selected classes,
3. selected member functions of selected classes, or
4. all member functions of all/selected parsed classes (i.e., classes for which static information has been generated).

Event trace information is generated when a breakpoint is hit or an exception is thrown. JDebugger keeps a stack of previously activated class members. The stack information can be used to determine the senders of the events. If the sender and the receiver of an event are the same, an action box is generated to the SCED scenario diagram. Otherwise, an event arc is generated. After the event or the action box is created, the execution of the program continues automatically. A hit of a breakpoint causes only a short pause of the execution of the system, giving the user no possibility to decide about the continuation of the execution.

Information about thrown exceptions is also added to the event trace. The user can choose a set of exceptions to be tracked using a menu command in Shimba. If the thrown exception was selected by the user, then the receiver for the event is set to be the class the exception is an instance of. Otherwise, the receiver is be the superclass of all exceptions, namely *java.lang.Exception*. If the user has not selected any exceptions, the *java.lang.Exception* class is the receiver for all exception events.

8.4.2 The control flow

Dynamic control flows of classes and/or methods can be extracted using breakpoints. In Java, branching of the execution is caused by conditions of **if**, **for**, **while**, **do-while**, and **switch-case** statements. JDebugger can be given instructions to set breakpoints at specific lines of classes. Hence, setting breakpoints on those conditional structures requires information about line numbers. This piece of information can be read from the byte code by JExtractor, as long as debugging information has been included to the class files. In jdk1.2, for example, this means that option “-g” or “-g:lines” has been used when compiling the source code.

A breakpoint that is set at a line of the actual condition that causes branching in the execution is called a *state breakpoint*. Correspondingly, an *assertion breakpoint* denotes a breakpoint at a line that might be executed right after the evaluation of the condition, that is, the line that is executed if the condition yields true/false or the first line of a chosen **case** branch of a **switch** statement. A hit of a state breakpoint causes a SCED state box to be added to the event trace. Correspondingly, a hit of an assertion breakpoint might cause a SCED assertion box to be added to the event trace. This approach causes a lot of breakpoints to be added, possibly even several for a single line. Adding only assertion breakpoints is not enough, since it is not always possible to determine which conditional statement caused a jump to a specific line. Hence, adding both assertion and state breakpoints is necessary. If an assertion breakpoint is hit, JDebugger needs to check which was the previous state box added to the scenario. An assertion box is added to the scenario only if the name of the previously added state box refers to the same line (e.g., a state box *TEST line 24* and an assertion box *Cond at 24 taken*). Note that the line number of a single condition of a conditional statement in Java byte code is uniquely determined by the line number of the operator used. When a state breakpoint is hit, JDebugger needs to check if there are also assertion breakpoints at that line. All such assertion breakpoints need to be checked first since the line might be also the first line in a branch of a previously executed conditional statement.

Note that Shimba is independent of the source code of the target software. Hence, state boxes are not labeled by the actual conditional structures, even though that would be more descriptive. If the source code existed, that would be possible since the line numbers are known. There are three major reasons for the chosen labels for state boxes and assertion boxes. First, we want to be totally independent of the source code. Second, since we do not give the condition written in the source code, we at least want to provide the user a possibility to quickly find that line if she wants to and she has an access to the source code files. Third, it does not seem to be more useful nor more descriptive to add the name of the actual byte code instructions that caused the branching. These byte code instructions are:

1 ifeq,

2 ifne,

3 iflt,

4 ifle,

5 ifgt,

6 ifge,

7 if_icmeq,

8 if_icmifne,

9 if_icmiflt,

10 if_icmifle,

11 if_icmifgt,

12 if_icmifge,

13 lcmp,

14 fcmpg,

15 fcmpl,

16 dcmpg,

17 dcmpl,

18 ifnull,

19 ifnonnull,

20 ifacmpeq,

21 ifacmpne,

22 goto,

23 goto_w,

24 lookupswitch, and

25 tableswitch.

From now on this instruction set is denoted by *I*. Consider the following fragment of Java source code:

```
22      public void readFromFile(String str) {
23          DataInputStream dataInputStream = getStream(str);
24          if (dataInputStream != null) {
25              read(dataInputStream);
26          } else {
27              printError("Stream not found");
28          }
29      }
```

After compilation the class file contains the following byte code instructions that correspond to the *readFromFile* method:

```
Method void readFromFile(java.lang.String)
    0 aload_0
    1 aload_1
    2 invokespecial ♯ 20 <Method java.io.DataInputStream getStream(java.lang.String)>
    5 astore_2
    6 aload_2
    7 ifnull 18
    10 aload_0
    11 aload_2
    12 invokespecial ♯ 25 <Method void read(java.io.DataInputStream)>
    15 goto 24
    18 aload_0
    19 ldc ♯ 2 <String "Stream not found">
    21 invokespecial ♯ 24 <Method void printError(java.lang.String)>
    24 return
```

The information about byte code instructions, as well as the following line number table, are generated by jdk's *javap* program.

Line numbers for the *readFromFile(java.lang.String)* method are:

line 23: 0

line 24: 6

line 25: 10

line 24: 15

line 27: 18

line 22: 24 (the method header)

According to the byte code instructions of the method *readFromFile* and the line number table, the first instruction that is executed at line 24 is *a_load* (number 6). The instruction *ifnull 18* (number 7) is executed after that. Branching is indicated with the number 18 after the name of the instruction. Thus, the next instruction to be executed is either number 10 or number 18, depending on the result of the test *ifnull*. Figure 8.2 shows a scenario diagram that has been generated for the target system. Before running the system, a breakpoint is set at line 24 because $ifnull \in I$. The JDebugger tool generates a message (using a call stack that is provided by the jdk debugger) when it stops at the first line of the method. It also stops at line 24 and adds the state box *TEST line 24* to the scenario diagram. The assertion *Cond at 24 taken* is added when the the execution has reached line 27.

```
public void initValue() {
    if (value == null) {
        // do nothing
    } else {
        value = new Integer(0);
    }
}
```

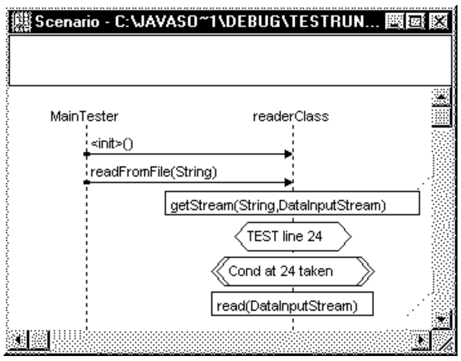


Figure 8.2: A scenario diagram describing an example run through the method *readFromFile(String)*

The byte code instructions for the *initValue* methods and the corresponding line numbers are:

Method void initValue()

```
0 aload_0
1 getfield ♯ 28 <Field java.lang.Integer value>
4 ifnull 19
7 aload_0
8 new ♯ 8 <Class java.lang.Integer>
11 dup
12 iconst_0
13 invokespecial ♯ 14 <Method java.lang.Integer(int)>
16 putfield ♯ 28 <Field java.lang.Integer value>
19 return
```

Line numbers for the *initValue()* method are:

```
line 32: 0
line 35: 7
```

line 31: 19 (the method header)

Note that in both cases the byte code instruction **ifnull** is generated because of the **if** statement even though the comparison operation in the first case is “!=” and in the second case “==”. Hence there is no way to know, even if the target system were compiled using the same Java compiler, which byte code instruction is generated from the conditional statement.

Information about the control flow is also needed for calculating some object-oriented metrics values (e.g., McCabe’s cyclomatic complexity [70]) [113, 114]). A flow graph is formed for each member function during the extraction of the static information (dependency graph). Instructions *goto* and *goto_w* of I have a specific role since they do not represent actual branching. They force the execution to jump to a given instruction. They are usually used in connection with other instructions in the list.

The algorithm for constructing a static control flow graph is presented next. The algorithm takes an ordered list of byte code instructions as input. An *offset number* is attached to each byte code instruction in the byte code stream. The offset number shows the offset in bytes from the beginning of the method’s byte code array to the start of the byte code instruction. The list of byte code instructions is ordered according to their offset numbers. *BinaryClass* and *BinaryConstantPool* classes of *sun.tools.java* package are used to get the byte code instruction array. For convenience, the set of conditional byte code instructions is split into three subsets:

1. I_1 = { *goto, goto_w* },
2. I_2 = { *lookupswitch, tableswitch* }, and
3. I_3 = $I - \{I_1 \cup I_2\}$.

In addition, six other instructions are considered. They all represent returns from the method: $I_4 = \{ireturn, lreturn, freturn, dreturn, areturn, return\}$.

Algorithm 3. Forming a control flow graph for a member function from Java byte code instructions.

Input: An ordered list *L* of byte code instructions.

Output: A control flow graph *CFG*.

Method:

```
formControlFlowGraph(L)
    $CFG := \emptyset$
    for each instruction l with offset number i in L do
        if $CFG$ contains a node with label $i$ then
            Let $x$ be that node.
        else
            Create node $x$ with label $i$ and add it to $CFG$.
        if l belongs to $I_I \cup I_3$ then
            Let $j$ be the instruction to be jumped to if the conditional byte code condition results
            true.
            if $CFG$ contains a node with label $j$ then
                Let $y$ be that node.
            else
                Create node $y$ with label $j$ and add it to $CFG$.
            Create a true edge from $x$ to $y$.
        else if l belongs to $I_4$
            // A sink node with id number 1000 is created for $CFG$.
            // The id number for the sink node needs to be greater
            // than any other id number of any other node in $CFG$,
            // i.e., it is the upper bound of the size of $CFG$.
            if $CFG$ contains a node with label 1000 then
                Let $sink$ be that node.
            else
                Create node $sink$ with label $j$ and add it to $CFG$.
            Create a *true* edge from $x$ to $sink$.
        **else if** *l* belongs to $I_2$
            Let $j$ be the offset number of the instruction representing the *default* branch.
            **if** $CFG$ contains a node with label $j$ **then**
                Let $y$ be that node.
            **else**
                Create node $y$ with label $j$ and add it to $CFG$.
            Create a *true* edge from $x$ to $y$.
            **for** each offset number $k$ that represents a *case* branch in the
            *tableswitch* or *lookupswitch* structure **begin**
                **if** $CFG$ contains a node with label $k$ **then**
                    Let $ns$ be that node.
                **else**
                    Create node $ns$ with label $k$ and add it to $CFG$.
                Create a *true* edge from $x$ to $ns$.
            **end** // for
        **end**
    **end** // for
    **if** the size of $CFG$ is greater that 1 **then**
        Set $prev$ to be the node in $CFG$ with the smallest label.
        Set $next$ to be the node in $CFG$ with the second smallest label.
        **while** $prev \neq$ **null begin**
            **if** $prev \notin I_1$ **then**
                Create a *false* edge from $prev$ to $next$.
            Let $prev$ be $next$.
            Let $next$ be a node in $CFG$ with the smallest label that is
            bigger than the current label of $next$.
        **end**
```

end

The Algorithm 3 does not produce a traditional control flow graph, formed from the source code and describing different paths through a method/module. However, according to the Theorem 8.4.1, their cyclomatic complexities, defined in Definition 8.4.1, are the same.

Definition 8.4.1 *The cyclomatic complexity ν of a graph G is*

$$\nu = e - n + p, \tag{8.1}$$

where e is the number of edges in G, n is the number of nodes in G, and p is the number of disconnected components in G.

Theorem 8.4.1 *For an arbitrary Java method M, let m_1 be the source code of M and m_2 the byte code of compiled m_1. Let CFG_s be a (traditional) control flow graph of m_1 and CFG_b a control flow graph generated by Algorithm 3 for m_2. Then the cyclomatic complexities of CFG_s and CFG_b are the same.*

Proof 8.4.1 *When generating CFG_b some extra nodes are added to the graph (e.g., defining the instruction to be jumped to). Those nodes do not necessarily belong to any of the sets I_1, I_2, I_3, or I_4. Let I_e be the set of those nodes.*

First, we show that for both graphs $p = 1$ (i.e., the graphs are connected). For CFG_s it is obvious. In CFG_b nodes that belong to $I_2 \cup I_3 \cup I_4 \cup I_e$ are connected in the order of their id number with false edges. Nodes in I_1 have always exactly one outgoing edge. They also have exactly one incoming edge, except possibly a node that has the smallest id number. If there were two components, the other component should only contain nodes that belong to I_1. That is impossible if m_2 is compiled from m_1.

It remains to be shown that $e - n$ has the same value in both graphs. Consider the following transformation of CFG_b: all nodes that have exactly one outgoing edge are removed and their *incoming edges are changed to enter the target nodes. Each time a node is removed, also an edge is removed, keeping the value of $e - n$ unchanged. In CFG_b such nodes are the ones belonging to I_1, I_4, or I_e. The nodes in I_3 always have two outgoing edges: one true edge representing the jump, and one false edge representing a continuation of the execution from the next instruction. Corresponding nodes are found from CFG_s as well. Such nodes are formed for all conditional statements (***if***, ***for***, ***while***, and ***do-while***). For all nodes in I_2, one true edge is generated to a node representing the default branch and to all case branches. Again, similar edges need to be found from CFG_s. Hence, the transformation results in a graph isomorphic to CFG_s, disregarding the labelling of nodes.*$\square$

As an example, consider the following method:

```
private void foo() {
    int val = 0;
    for (int i = 1; i < 4; i++) {
        for (int j = 1; j < 5; j++) {
            val += i*j;
        }
    }
    switch (value) {
    case 1:
        value = val + value;
        break;

    case 2:
        value = val - value;
    default:
        value = val;
    }
}
```

After compilation the class file contains the following byte code instructions that correspond to the method:

```
Method void foo()
   0 iconst_0
   1 istore_1
   2 iconst_1
   3 istore_2
   4 goto 29
   7 iconst_1
   8 istore_3
   9 goto 21
   12 iload_1
   13 iload_2
   14 iload_3
   15 imul
   16 iadd
   17 istore_1
   18 iinc 3 1
   21 iload_3
   22 iconst_5
   23 if_icmplt 12
   26 iinc 2 1
   29 iload_2
   30 iconst_4
   31 if_icmplt 7
   34 aload_0
```

```
35 getfield ♯ 8 <Field int value>
38 tableswitch 1 to 2: default=83
    1: 60
    2: 73
60 aload_0
61 iload_1
62 aload_0
63 getfield ♯ 8 <Field int value>
66 iadd
67 putfield ♯ 8 <Field int value>
70 goto 88
73 aload_0
74 iload_1
75 aload_0
76 getfield ♯ 8 <Field int value>
79 isub
80 putfield ♯ 8 <Field int value>
83 aload_0
84 iload_1
85 putfield ♯ 8 <Field int value>
88 return

Line numbers for method void foo()
  line 14: 0
  line 16: 2
  line 17: 7
  line 18: 12
  line 17: 18
  line 16: 26
```

line 22: 34

line 24: 60

line 25: 70

line 27: 73

line 29: 83

line 13: 88

The information about the byte code instructions, as well as the line number table, are generated by jdk's *javap* program. Figure 8.3 shows the generated control flow graph on the left, and the one formed from the source code on the right. The cyclomatic complexity of both graphs is the same. Application of the transformation method described in Proof 8.4.1 for nodes 4, 7, 9, 12, 21, 29, 70, and 88, results in a graph that is isomorphic to the graph on the right disregarding the labelling of the nodes.

8.5 Managing the explosion of the event trace

A large amount of event trace information is generated as a result of even a relatively brief usage of the target system. Thus, methods for managing the event explosion problem are needed. Searching behavioral patterns (i.e., repeated similar behavior) from the event trace provides one way to deal with the problem. This approach is used, for example, in ISVis [51] and Jinsight [45]. In Shimba, the original scenarios (or any subset of them) can be modified by applying string matching algorithms to them. The patterns found provide means to raise the level of abstraction of the scenarios and to decrease their size.

Shimba uses string matching algorithms to search behavioral patterns from SCED scenario files. A pattern consists of a sequence of any SCED scenario diagram items. Shimba recognizes exact matches only (i.e., both the name and the owner/owners of each scenario diagram item has to be the same). The behavioral patterns found are shown either as repetition constructs or as subscenarios. Repetition constructs are used if the pattern is repeated at least twice in a row. Patterns that occur in a row are potentially generated by *while*, *for*, or *do-while* structures. Hence, a repetition

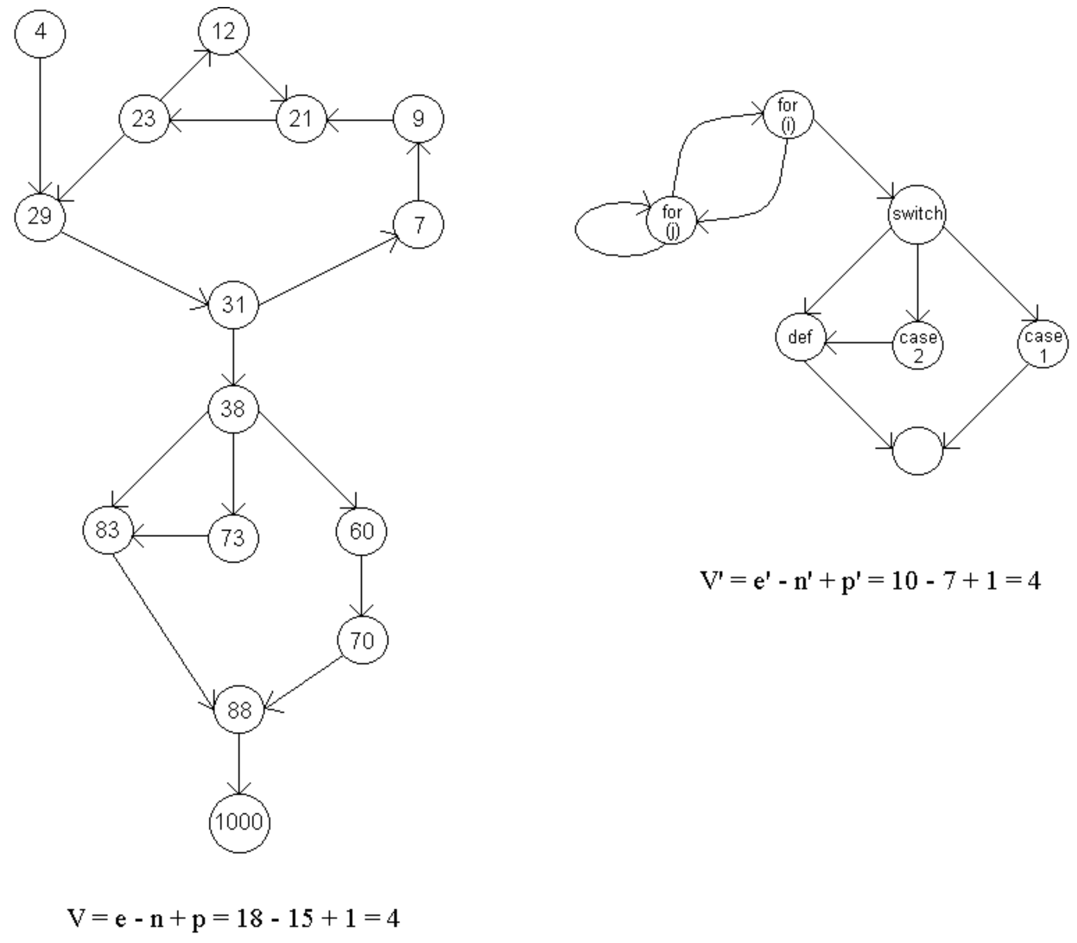

4
12
23
21
9
29
7
31
38
83
73
60
70
88
1000
V = e - n + p = 18 - 15 + 1 = 4
for (i)
for (j)
switch
def
case 2
case 1
V' = e' - n' + p' = 10 - 7 + 1 = 4

Figure 8.3: Two control flow graphs for the *foo* method; the one generated from the byte code on the left and the one formed based on the source code on the right. The cyclomatic complexity ν for both graphs is 4.

construct is a natural way to show the same situation in a scenario. The name of the repetition construct indicates the number of times the pattern has been repeated. Repeated events may appear also in other circumstances. For example, when debugging the behavior of a GUI of a software system, moving a mouse, clicking a mouse, keyboard commands, etc., often cause an event to be repeated several times in a row. The number of repetitions might be large, even compared to the size of the event trace. Hence repetition constructs for repeated single events are generated already during the debugging process (i.e., when generating the original scenario diagrams).

Behavioral patterns may also occur independent of each other. For example, in a GUI software, opening of a dialog box causes several events to be executed in a row (initialization of the dialog). Such a pattern is repeated every time a dialog is opened. These patterns may appear in different scenarios, or at least, disconnected from each other. Subscenarios are used to identify such patterns. Subscenario boxes provide a powerful way to link scenario diagrams: instead of repeating the event sequence in a place of their appearance, a subscenario box is used to refer to another scenario diagram that actually contains the event sequence. When double clicking a subscenario box, the scenario diagram containing the detailed pattern information is opened. Subscenarios help the engineer to recognize the patterns, to browse them, and to structure the described behavior. The synthesis algorithm reads the events of nested scenario diagrams recursively. The resulting state diagram is similar to the one generated for the set of original, unnested scenario diagrams. The Boyer-Moore string matching algorithm [12] is applied for searching the behavioral patterns.

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

8.6 Merging dynamic information into a static view

In Shimba, dynamic event trace information can be attached to a static dependency graph. When merging dynamic information into a static view, 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 (cf. Appendix A), generated when an exception is thrown, and *throw* edges (cf. Appendix B) indicating which member functions threw the exception. The *throw* edges are naturally given weight values as well.

The merged view can be saved in an extended RSF format. An (unstructured) RSF file or stream consists of a sequence of triples, one triple on a line [123]. The triple format of RSF is insufficient for giving weight or any other kind of attribute values for edges. In this research, attributes for edges are given as lines that start with two comment characters (♯♯). For example, the following line defines the weight of a method call to be 8:

♯♯ *call uml.UMLIncrement.markEdited() uml.UMLMethod.setDirty() weight 8*

The resulting graph can be loaded into Rigi using a couple of simple scripts that are able to parse such comment lines. The standard *rcl_load_rsf* script of Rigi [123] is used to load the rest of the RSF file. Figure 9.17 shows a Rigi graph that contains both static and dynamic information.

8.7 Using static information to guide the generation of dynamic information

Using Shimba, static information can be generated before, during, or after debugging the target system. If the static dependency graph exists before generating the dynamic models, JDebugger can be given instructions based on the static information. Possibilities of such guidance are discussed next.

When reverse engineering an unknown system, the engineer might be interested in a specific part of the software. In static reverse engineering, the user can extract information for only those class files she is interested in. For example, JExtractor can be given instructions to generate a static dependency graph for a specific package only, if the class files reside in a directory structure that reflects the package names, which is common for Java software. In dynamic reverse engineering, if the user is interested in the dynamic behavior of a specific part of the software, it is not meaningful to generate information about all the object interactions of the system, but only the interactions that have an effect on the behavior of that part. Such information slicing can dramatically decrease the size of generated event traces, still containing the information of interest. In order to be able to do that, the engineer needs to know the dependencies among the classes. This means that knowledge about the static structure of the software is needed.

Rigi provides several scripts for analyzing the static (or merged) graph. In addition, new scripts can be easily written and added to the script library. In Shimba, the slicing mechanisms of Rigi can be used to guide JDebugger to restrict the amount of event trace information to be generated. After defining the parts of interest from a Rigi graph, the user can select all the *Method*, *Constructor*, and *Staticblock* nodes (representing member functions) for which JDebugger will set breakpoints.

Assume that the user is interested in the behavior of classes that belong to a set S_o. Let S be a set of classes such that $S_o \subseteq S$ and all inherited classes of each $s \in S_o$ belong to S. 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 meaningful then to filter out everything else from the Rigi graph except the following nodes:

1. all classes and interfaces in S,
2. all members in M,

3. all nodes in M_s,

4. all nodes in M_t, and

5. all nodes that represent overridden methods of methods in M_t.

The user can then generate dynamic information to the software artifacts that belong to the above set of nodes. This set includes all the software artifacts that might effect the behavior of S_o, hence guaranteeing that all the essential information will be included in the generated event trace.

The user may also find nodes that represent software artifacts outside the target system uninteresting. Such nodes might be, for example, methods and classes belonging to jdk.

Finding the right nodes using Rigi is simple, since the package and class declaration are included in the names of all nodes. For example, *Method* nodes are of type *packageName.className.methodName*. Hence, the nodes can be found using a simple grepping script.

8.8 Slicing a Rigi view using SCED scenarios

The dynamic information can be used for analyzing the static dependency graph of a subject system. Section 8.6 discussed how dynamic information can be attached to a static Rigi graph. The graph can then be sliced based on this dynamic information. For example, parts of the software that has not been used (heavily enough) can be filtered out. In this section we discuss how a purely static graph can be sliced based on dynamic event trace information. By a slice of the Rigi graph we mean a subgraph that shows only desired parts of the original Rigi graph.

When reverse engineering SCED scenario diagrams the senders and receivers of member function calls can be set to be either objects or classes. Thus, by examining the SCED scenarios a called member function can be identified but the caller is given as an object or a class, although the member function is always called from another member function. The generated static Rigi graph also shows a *call* relationship between two member functions. Such relationships can be given weight values indicating their actual run-time usage (cf. Section 9.17). A Rigi graph that contains both static and dynamic information can be sliced based on the weight values. For example, the user can filter out parts of the graph that have not been used (heavily enough) at run-time.

Slicing the purely static Rigi graph based on example scenarios differs from slicing a Rigi graph, in which dynamic information has been merged. In the former case Method, Constructor, Staticblock, and Class nodes that 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. The slicing is performed by Algorithm 4 and Algorithm 5. The algorithms have been implemented in Tcl/Tk and can be run from Rigi.

Algorithm 4. Slicing a purely static Rigi graph based on a set of SCED scenario diagrams.
Input: A set S of SCED scenarios and a Rigi graph G generated from Java byte code.
Output: A sliced Rigi graph *G'*.
Method:

```
sliceByScenarios($S$, $G$)
    Deselect all nodes in $G$.
    for each scenario $s \in S$
        selectByAScenario(s, G)
    Reverse the selection in $G$.
    Form a sliced graph $G'$ by filtering out all selected nodes in $G$.
    Return $G'$.
end
```

Algorithm 5. Selecting nodes in a Rigi graph based on a single SCED scenario diagram.
Input: A SCED scenario diagram S and a Rigi graph G generated from Java byte code.
Method:

```
selectByAScenario($S$, $G$)
```

```
    for each scenario item $item \in S$ begin
        if $item$ is a member function call then
            Select a Method, a Constructor, or a Staticblock node in $G$ that represents the called
            member function.
            Select a Class node in $G$ that represents the sender of $item$.
            Select a Class node in $G$ that represents the receiver of $item$.
        if $item$ is an action box then
            Select a Method, a Constructor, or a Staticblock node in $G$ that represents the called
            member function.
            Select a Class node in $G$ that represents the owner of $item$.
        if $item$ is a subscenario then
            selectByAScenario($item$, $G$)
    end // for
end
```

The presented dynamic slicing approach can be used for finding structural flaws in the software that cause unexpected or undesired behavior. For example, a bug in the software might cause a failure in the execution inevitably or only with a certain input or when the software has been used in a specific way. By examining the parts of the software that are involved in a usage that causes the failure the user might be able to conclude whether the failure was inevitable or not. Section 9.4.2 discusses a realistic example of a such case. In general, the dynamic slicing approach can be used for studying the structure of parts of the target software that are involved in a specific kind of usage.

8.9 Raising the level of abstraction of SCED scenarios using a high-level Rigi graph

In the previous section we demonstrated how a Rigi graph can be modified by a set of SCED scenarios. In this section a method of modifying a set of SCED scenario diagrams by a high-level

Rigi graph is presented. Section 9.4.3 discusses an example of using the method.

The dynamic reverse engineering process results in many low-level SCED scenario diagrams. Hence, means to raise the level of abstraction of the scenarios are needed. Usually this is done by searching for behavioral patterns. In some cases, however, it is meaningful to build abstractions for the scenario diagrams based on static criteria. The original scenario diagrams show the interaction among objects or classes. From those diagrams it is difficult to understand the interactions among high-level static components (e.g., between Java packages), which might be useful to get a flavor of the overall communication within the target system.

Rigi provides several algorithms for building abstractions for the initial static dependency graph. In addition, the user can easily write and run new ones and add them to the script library. Subsystems can be composed by taking advantage of some metrics values. A commonly used heuristic for finding potential subsystems is "high cohesion within subsystems and low coupling among subsystems". A subsystem is highly cohesive if its subparts have a lot of dependencies with each other. The part has low coupling if the subparts have only few dependencies with elements outside that part.

Object-oriented languages, and especially pure object-oriented languages (e.g., Java), provide extended ways to build the abstractions since they support encapsulation. Such language structures can also be used to build the abstractions automatically. For example, examining Java software by observing classes and their relations might clarify the overall structure of the software, compared to studying it at the level of object interactions. In Rigi such abstractions can be built by collapsing all object nodes representing instances of a single class into one class node, hence making the graph considerably smaller. Examining the structure at the class level might still contain too detailed information. The next step could be to collapse all classes and interfaces into packages, etc.

In Shimba, the information about abstractions in a Rigi graph can be used to build abstractions for

SCED scenario diagrams. Consider a high-level Rigi graph constructed for a target Java software system, and a SCED scenario diagram S generated when running the same software. Let p be a participant in S, labeled Cl. If p does not represent an exception or a participant outside the target system, and there is no node in the Rigi graph with label Cl, then a node with label Cl must have been collapsed into a high-level node in Rigi. Let that high-level node have a name Cl_a. In the used method SCED scenario diagram participant p is replaced by a new participant with label Cl_a. Because nodes can be nested to an arbitrary depth, the name of the highest-level node is chosen. 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 member function that has been called. For example, if the original scenario included an event *m* from *package1.class1* to *package2.class2*, the changed scenario might include an event *class2.m* from *package1* to *package2*. All action boxes are removed from the changed scenarios, as well as events that would be represented as action boxes after the conversion (i.e., the sender and the receiver of the event are the same). By removing all action boxes the scenarios become much shorter and the interaction among the high-level participants is emphasized. Furthermore, all the repeat blocks that become empty are removed. Finally, if a scenario diagram contains subscenario references, the referred subscenarios are changed, too.

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. The abstraction in that case has probably been built by collapsing all participants representing instances of a class into single vertical line that represents the class. Vertical abstractions decrease the number of horizontal arcs in a scenario. They can be built, for example, by omitting “internal” calls of a single participant or by collapsing method call chains into a single call event. The approach introduced above can be used to build both horizontal and vertical abstractions. The static abstractions built 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. Vertical abstractions are produced automatically by omitting internal method calls.

8.10 Related work

Several tools and environments supporting reverse engineering and architecture recovery rely on static analysis of the software, for example, Rigi [74], Bookshelf [34], CIAO [15], and Sniff+ [115]. In addition, many tools supporting forward engineering of object-oriented software are also able to extract class diagrams for existing software. We consider here tools, environments, and methods that

1. support dynamic reverse engineering of object-oriented software systems by constructing dynamic views to the target software, or

2. aim at combining static and dynamic information for constructing views of the target software.

8.10.1 Dynamic reverse engineering tools

In Ovation, *execution pattern views* are used to visualize and explore a program's execution at different levels of abstraction [26, 27]. An execution pattern is derived from Jacobson's interaction diagrams [50] and is represented in a tree structure. Ovation offers several means to manipulate the view. For raising the level of abstraction and for dealing with the event explosion problem, the views can be flattened, subtrees can be collapsed, (nested) repetition constructs can be created, etc. Repetition constructs can be built for similar execution patterns. As in our approach, repetition constructs are used for viewing execution patterns in Ovation. The execution patterns do not, however, always occur in succession. In fact, the patterns that are most difficult to find and are often most interesting ones, do not occur in succession. The subscenario constructs of SCED can be used to view those patterns and to structure the set of scenario diagrams generated. Ovation does not include such a concept. Ovation offers the user a possibility to define the similarity of patterns. She can choose one or more criteria from eight categorized ones, to be used by the pattern matching algorithms to decide if two pattern are "similar" or not. In our approach, behavioral patterns can be searched based on exact string-based matches only.

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 [99]. 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 information can then be shown using different views. The fine-grained object interaction is shown as *Object Interaction Diagrams*, class and framework interaction is shown as *Affinity Diagrams*, and framework and subsystem interaction is shown as *Ternary Diagrams*. The Object Interaction Diagram is a variation of an MSC, and the Affinity Diagram is a directed graph. 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. If she wants to view more or less detailed information, she has to run the software again. In other words, there is no information exchange among different views. In our approach, state diagrams can be synthesized from a set of scenario diagrams, thus providing the user a possibility to examine the information extracted using two different views. On the other hand, compared to our approach, the architectural-aware instrumentation mechanism is more efficient in data gathering. The approach described by Sefika *et al.* only supports dynamic reverse engineering. Hence, it might be difficult for the user to relate the run-time entities to the source code artifacts.

Walker *et al.* use high-level models for visualizing program execution information [120]. 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 scenario views using a mapping between low-level software artifacts and high-level components [112].

Jinsight is a tool for visualizing the dynamic behavior of Java programs [45]. It views information about object population, method invocations, garbage collection, CPU and memory bottlenecks, thread interactions, and deadlocks. The event traces are produced as a result of instrumenting Java Virtual Machine. Jinsight offers several kinds of views that can be used to analyze the information captured in event traces. The overall object interaction can be examined using *Histogram Views*, a *Call Stack View* shows the call stack for each thread, and references between objects can be viewed using a *Reference Pattern View*. Furthermore, Jinsight uses several views for showing event traces over time: an *Execution View* shows an overview of communication among objects per thread, while an *Invocation Browser View* and *Execution Pattern Views* can be used for browsing pieces of event traces. The Invocation Browser View shows method calls and messages to a selected object, as well as all subsequent communication and the Execution Pattern view shows a summary of recurring interaction patterns arising from the selected method [45]. Jinsight uses a tree structured interaction diagram notation that resembles execution pattern views of Ovation [27, 26]. Jinsight is one of the most versatile dynamic reverse engineering tools available. It includes various views for examining different aspects of the run-time behavior. However, state machines, which are commonly used diagrams for specification of the dynamic behavior, are not supported by Jinsight. In addition, Jinsight does not generate information about the dynamic control flow of a selected object or method. These features are included in Shimba. Jinsight does not support static reverse engineering.

A source code instrumentation technique is used in Scene for producing event traces and visualizing them as scenario diagrams [59]. Scene allows the user to browse not only scenarios but also various kinds of associated documents, such as source code, class interfaces, class diagrams, and call matrices. For compressing the large amount of extracted event trace information Scene shows the operation calls (messages) in a closed form as default: the internal events of a call are not shown unless 'opened' by clicking the call arc. In this way the user can proceed to the interesting level, in a top-down fashion. Corresponding "horizontal abstractions" can be made using execution pattern views of Ovation [27, 26]. Scene offers other means to further narrow the set of displayed calls. If the number of participants becomes too large, a new scenario window is automatically opened for a call. Scene does not take advance of behavioral patterns for structuring and decreasing the information shown in scenario diagrams. Such a facility in provided in Shimba. As the event trace compressing method of Scene, the behavioral patterns shorten the event trace (i.e., decreases the number of scenario items). In Shimba, the event trace information can be compressed also vertically (i.e., to decrease the number of participants in a scenario diagram) using high-level components constructed in Rigi. As Jinsight [45], Scene does not support static reverse engineering, which would again provide a way to decrease the huge amount of event trace information. Scene is implemented in and for the Oberon environment.

8.10.2 Tools that combine static and dynamic information

ISVis is a visualization tool that supports the browsing and analysis of execution scenarios [51]. A source code instrumentation technique is used to produce the execution scenarios. ISVis belongs to the MORALE tool set, which aims at facilitating the evolution of legacy software systems [89]. To avoid event explosion, the user can select 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. A corresponding method in our approach is described in Section 8.9. 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.

Program Explorer combines static information with run-time information to produce views that summarize relevant computations of the target system [60, 61]. 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.

Richner and Ducasse introduce a query-based approach to recover high-level views of object-oriented applications is presented [87]. 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 among different views.

Dali is a workbench for architectural extraction, manipulation, and conformance testing [52]. It integrates several analysis tools and saves the extracted information 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.

8.11 Summary

The ultimate goal in software construction is to build software systems that provide desired behavior. The behavior of a program is important to end-users, but they are usually not interested in its structure. The execution breathes life into a software system. On the other hand, the structure of the software system defines its behavior. A program can be written without executing it but it cannot have a behavior before it is constructed.

Because of the strong bond and dependence between structural and behavioral aspect of the software, the static and dynamic analysis of the software should also be coupled. This in not supported well in currently available reverse engineering tools. Many of the tools are focused on either static or dynamic reverse engineering but not both. The tools that support both static and dynamic reverse engineering usually either merge the extracted information into a single view or construct and analyze static and dynamic views separately. Neither of these approaches takes full advantage of the information extracted.

Shimba combines static and reverse analyses at an early stage of the reverse engineering process. The static Rigi views, for instance, can be used to guide the generation of dynamic information. Furthermore, Shimba allows information exchange between the static and the dynamic views. The overlapping information of the views provides a channel which enables that. Static and dynamic views can thus be used to improve and modify each other. For example, the proposed techniques can be used to slice a view with another view and to build high-level views using other views.

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

1. What are the static software artifacts and how are they related?

2. How are the software artifacts used at the run-time?

3. What is the high-level structure of the software?
4. How do the high-level components interact with each other?
5. Does the run-time behavior contain behavioral patterns that are repeated? If it does, what are the patterns and in which circumstance do they occur?
6. How heavily has each part of the software been used at run-time and which parts have not been used at all?

Goal-driven reverse engineering approaches are useful, for example, for debugging. Tracking down a bug might be difficult. For example, consider a software system that is irregularly unstable. In this case, it might not be sufficient to know when the failure occurs, or even what has happened before the failure. The engineer needs to find out in which order these things occurred before the failure. Shimba supports debugging by offering automated tools that can be used to answer the following questions concerning exceptional behavior of the subject software system:

1. How does a certain part of the software behave?
2. When were exceptions thrown ? What happened before they were thrown and in which order ? What are the exceptions and who threw them?
3. When did an error occur, what happened before the failure and in which order?
4. How is the part that causes exceptional behavior constructed?

In Shimba, a dynamic control flow graph, that is, a dynamic slice of the implicit (static) control flow, can be synthesized automatically for desired objects or methods and visualized as a state diagram. Dynamic control flows are useful for detecting decision making during the execution, for profiling, for studying the cyclomatic complexity, for investigating code coverage, etc. Since the dynamic control flow is constructed automatically based on the usage of the target software, the user can generate desired dynamic slices of the implicit control flow.

Even a relatively brief usage of the subject software system typically produces a large amount of event trace information. In Shimba, the event trace information is stored into a set of SCED scenario diagrams. Some dynamic reverse engineering tools provide means to search behavioral patterns from the event trace information [26, 51, 27]. This is also supported in Shimba. However, it is difficult to understand the overall behavior of a single object by examining the scenario diagrams or the behavioral patterns. The state diagram synthesis technique of SCED helps the engineer to achieve this task. Shimba thus 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. To the best of our knowledge, similar features are not provided by other dynamic reverse engineering tools.

In addition to debugging, Shimba support other goal-driven reverse engineering tasks. For example, the following task specific question can be answered:

1. What is the dynamic control flow and the overall behavior of an object or a method?
2. How can a certain state in the object's life 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 the execution?
5. Which part of the target software has dependencies with this object or method (or any predefined part of the software)?

Chapter 9

A case study: reverse engineering FUJABA software

To validate the usefulness of the approach explained in Chapter 8, a target Java software system was examined. In this chapter we present the results of that case study, which was carried out by the author. Section 9.1 characterizes the purpose of the case study. The target software system in introduced in Section 9.2. Section 9.3 discusses several examples of reverse engineering the dynamic behavior of selected parts of the target software. In Section 9.4 the static and the dynamic models are used to complement each other in ways presented in Sections 8.7, 8.8, and 8.9. A summary of the case study is given in Section 9.5.

9.1 Tasks

In this case study we examine both static and dynamic aspects of a target Java software system FUJABA [88], the overall focus in the case study being on dynamic reverse engineering. The purpose of the case study is to test and validate the reverse engineering techniques provided by Shimba. Shimba is used for both general program understanding purposes and for goal-driven reverse engineering tasks.

To understand the overall structure of FUJABA, static information is generated for the whole software and visualized using Rigi. Limiting to a specific part of the target software system is problematic, since sometimes the engineer can understand the structure of the part only if she knows how it is related to the rest of the system. Furthermore, we aim to make several queries on the static dependency graph for various goal-driven reverse engineering tasks. We also want to build static abstractions and slice the Rigi graph with a set of scenario diagrams. These tasks would be difficult or even impossible to achieve, if the static dependency graph did not cover the whole FUJABA software.

In this case study, several dynamic reverse engineering tasks are set. Most of these tasks are goal-driven. We use Shimba to analyze the behavior of specific objects and methods. When studying the dynamic usage of a single method, we seek answers to questions of the following form:

1. What are the methods that have call dependencies with this method?
2. What is the overall run-time usage of the method?
3. What is the dynamic control flow of the method?
4. Was the run-time usage diversified enough to produce information that covers all possible use cases?

Similar questions are asked when the behavior of an object is studied. The dynamic analysis of FUJABA is not limited to examining the behavior of single objects or methods. We also aim to visualize the dynamic information using high-level views.

Debugging typically starts by identifying the exceptional behavior. When the point of failure is found, the behavior is analyzed further and the source code is examined to find the source of the failure. Shimba provides graphical support for debugging. In this case study we analyze a known bug in FUJABA. We first use the dynamic models to analyze the exceptional behavior and then examine the Rigi graph to study the structure of the part of FUJABA that caused the failure.

Shimba stresses the importance of combining static and dynamic reverse engineering, providing several techniques to exchange information between the models and to modify one model based on the information given by another model. In this case study, we test the usefulness of these techniques by asking the following questions:

1. How does a certain part of the software behave?
2. What kind of high-level components can be built for the static dependency graph?
3. How do high-level static components interact with each other?
4. How heavily has each part of the software been used at run-time?
5. Which parts of FUJABA are needed for a specific behavior and how these parts have been constructed?

9.2 The target Java software: FUJABA

The selected target system FUJABA [88] was developed at the University of Paderborn, Germany, and is freely available. 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 (class diagrams) and behavioral (activity diagrams, UML activity/story diagrams) aspects of a software. Furthermore, the Java source code can be generated from the design, which then can be compiled. The FUJABA environment also supports animation of the designed system through the constructed models. FUJABA is written in Java, consisting of almost 700 classes. The FUJABA version under examination is 0.6.3-0.

Figure 9.1 shows the initial graph loaded into Rigi from a file generated by the byte code extractor written by the author for the whole FUJABA software. The nodes in the static dependency graph represent software artifacts belonging to FUJABA. In addition, the graph contains nodes that represent software artifacts needed by FUJABA. Such artifacts typically belong to jdk or a library used by FUJABA. The arcs in the graph describe dependencies among the artifacts. The bottom left corner indicates that the static dependency graph consists of 25854 software artifacts.

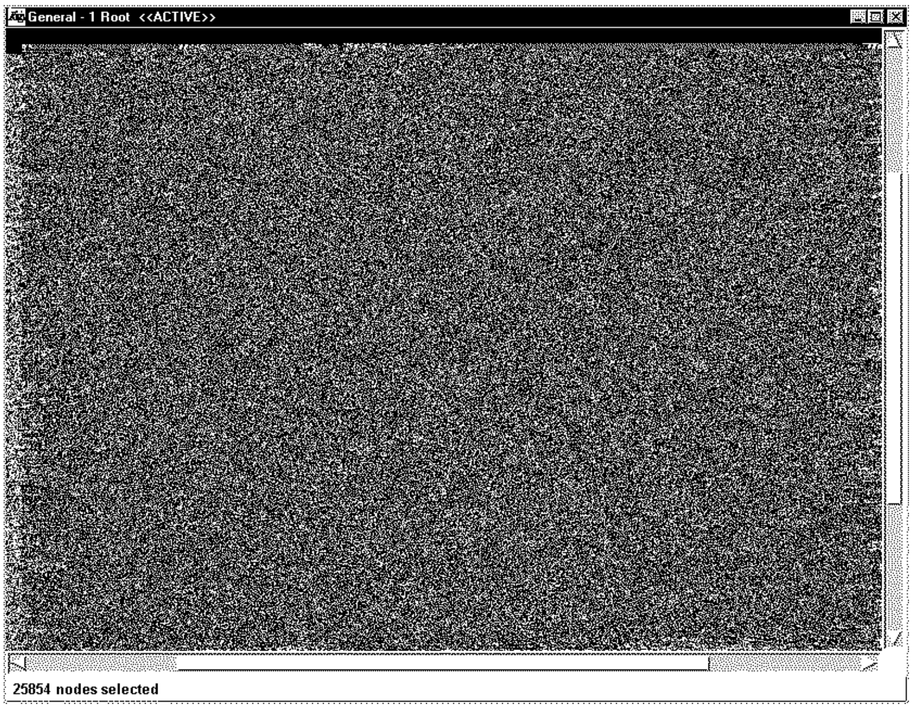


Figure 9.1: The initial Rigi graph representing the whole FUJABA software

9.3 Dynamic modeling

Next we focus on studying the functionality of the class diagram editor of FUJABA. In the following sections examples of constructing dynamic models are presented. The internal behavior and the dynamic control flow of a single method is shown as a SCED state diagram. The state diagram is synthesized automatically from the scenarios generated when running FUJABA under JDebugger. State diagrams are also used for modeling the overall behavior of two different objects. In addition, SCED scenario diagrams are structured and reorganized by behavioral patterns that are found using string matching algorithms. Finally, SCED scenario diagrams are used to trace the source of a known bug in FUJABA.

9.3.1 Modeling the internal behavior of a method

Figure 9.2 shows a dialog box used in FUJABA for defining and editing parameters of methods. In this section we focus on examining the internal behavior and the run-time control flow of the *mod-*

ify button. Each time the *modify* button is pressed, method *modifyButton_actionPerformed(ActionEvent)* of the dialog class *PEParameters* is called.

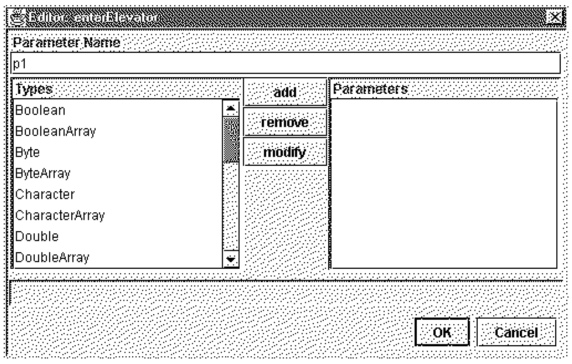


Figure 9.2: A dialog used in FUJABA class diagram for defining and editing parameters for methods

By running few scripts in Rigi, the method *modifyButton_actionPerformed (ActionEvent)* of class *de.uni_paderborn.fujaba.gui.PEParameters*, nodes that depend on it, and nodes that are referenced by it can be easily separated from the rest of the system. Furthermore, it can be easily certified (by running a couple more scripts) that none of the methods that are called by the method *modifyButton_actionPerformed (ActionEvent)* is overridden in any subclasses. Figure 9.3 shows a Rigi graph, in which all the other nodes have 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 in understanding the behavior of the target system itself. In addition, tracking down the invocations of methods belonging to packages outside the target system would increase the size of the event trace dramatically, containing often useless information. For example, methods of the class *java.lang.String* are called frequently but such calls are meaningless for understanding, for example, the behavior of the dialog in Figure 9.2. However, information about the usage of packages that provide functionality of an object-oriented framework would be interesting in many cases.

Breakpoints were set at the first line of all member functions chosen from the Rigi graph in Fig-

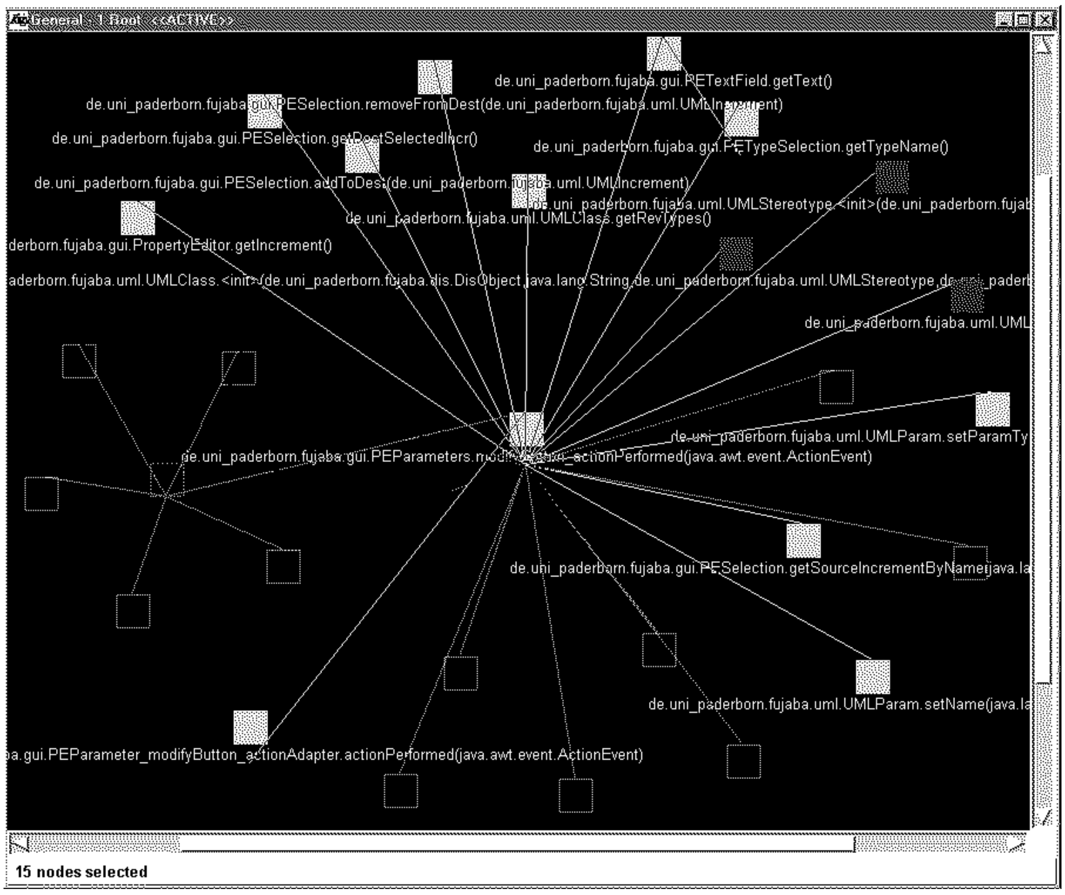

General - 1 Root <<ACTIVE>>
de.uni_paderborn.fujaba.gui.PETextField.getText()
de.uni_paderborn.fujaba.gui.PESelection.removeFromDest(de.uni_paderborn.fujaba.uml.UMLIncrement)
de.uni_paderborn.fujaba.gui.PESelection.getDestSelectedIncr()
de.uni_paderborn.fujaba.gui.PETypeSelection.getTypeName()
de.uni_paderborn.fujaba.gui.PESelection.addToDest(de.uni_paderborn.fujaba.uml.UMLIncrement)
de.uni_paderborn.fujaba.uml.UMLStereotype.<init>(de.uni_paderborn.fujab
de.uni_paderborn.fujaba.uml.UMLClass.getRevTypes()
derborn.fujaba.gui.PropertyEditor.getIncrement()
aderborn.fujaba.uml.UMLClass.<init>(de.uni_paderborn.fujaba.dis.DisObject,java.lang.String,de.uni_paderborn.fujaba.uml.UMLStereotype,de.uni_padert
de.uni_paderborn.fujaba.uml.UML
de.uni_paderborn.fujaba.uml.UMLParam.setParamTy
de.uni_paderborn.fujaba.gui.PEParameters.modifyButton_actionPerformed(java.awt.event.ActionEvent)
de.uni_paderborn.fujaba.gui.PESelection.getSourceIncrementByName(java.la
de.uni_paderborn.fujaba.uml.UMLParam.setName(java.la
a.gui.PEParameter_modifyButton_actionAdapter.actionPerformed(java.awt.event.ActionEvent)
15 nodes selected

Figure 9.3: A Rigi graph, in which nodes represent software artifacts and edges are relationships among them. Method node *modifyButton_actionPerformed(ActionEvent)* of class *de.uni_paderborn.fujaba.gui.PEParameters* is shown in the center of the graph. The other nodes in the graph are dependent on artifacts of that method, from which 11 method nodes and 3 constructor nodes are selected.

ure 9.3 (in this case 11 methods and 3 constructors). In addition, in order to capture branching in the execution state breakpoints and assertion breakpoints were set in the method *modifyButton_actionPerformed(ActionEvent)*. JDebugger does not normally create events corresponding to returns from methods, only the method calls are recorded. However, the state diagram synthesis algorithm for methods reads all the scenario items between the method call and the corresponding return event. Therefore, breakpoints were also set for all the *return* statements of method *modifyButton_actionPerformed(ActionEvent)*.

To capture the behavior of the *modify* button, the dialog in Figure 9.2 was used 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 was selected.
3. The type of the selected parameter was changed and the *modify* button was pressed.
4. The name of the selected parameter was deleted and the *modify* button was pressed.

The scenarios resulting from the usage of the dialog in the first and third cases turned out to be similar. We selected the second and fourth cases for the case study since they represent a slightly exceptional usage of the dialog. Figure 9.4 shows the scenario resulting from the first case, and Figure 9.5 shows the merged internal behavior of the method *modifyButton_ actionPerformed(ActionEvent)* of class *de.uni_paderborn.fujaba.gui.PEParameters*, synthesized from four scenarios. For a comparison, the actual source code is shown in Figure 9.6.

The implicit (static) control flow for a method can be constructed from the source code (or its byte code) of a target software system. Figure 9.5 shows the dynamic control flow of method *modifyButton_actionPerformed(ActionEvent)*, that is, a dynamic slice of the implicit (static) control flow. A state diagram is a powerful and natural graphical representation to examine the dynamic

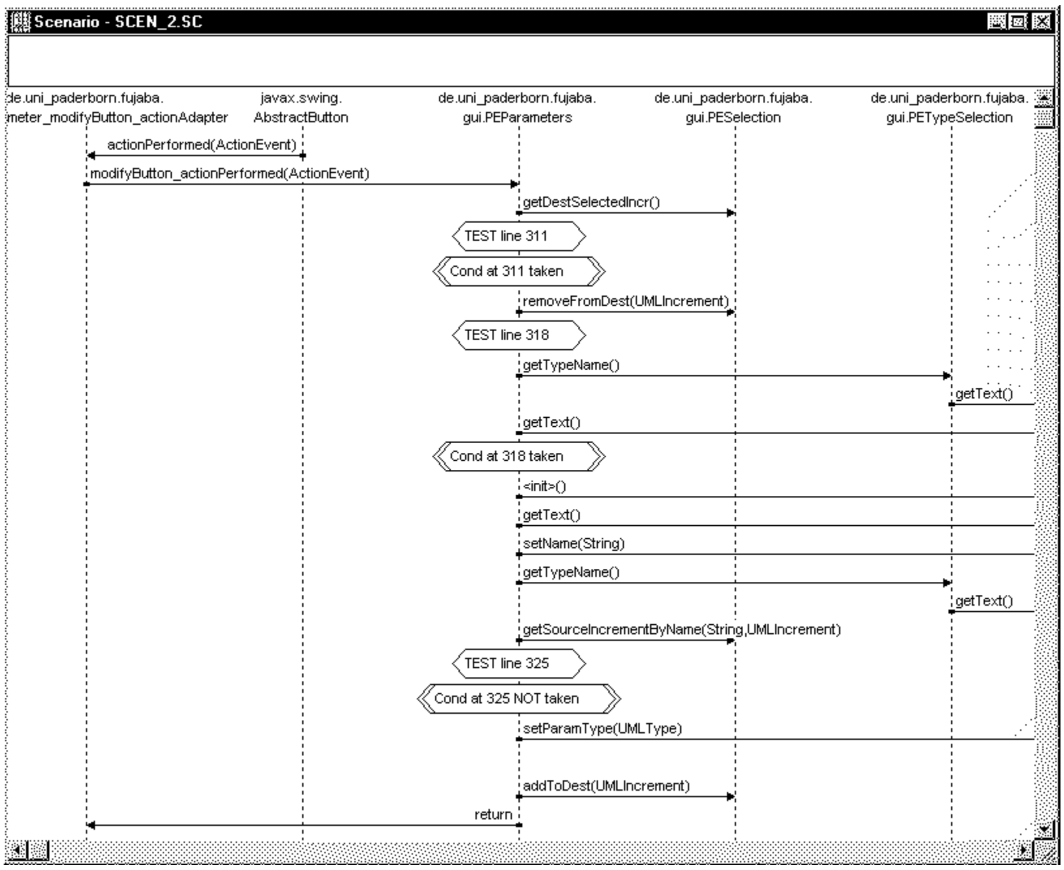


Figure 9.4: A scenario describing one execution path through the method *modifyButton_actionPerformed(ActionEvent)* of class *de.uni_paderborn.fujaba.gui.PEParameters*

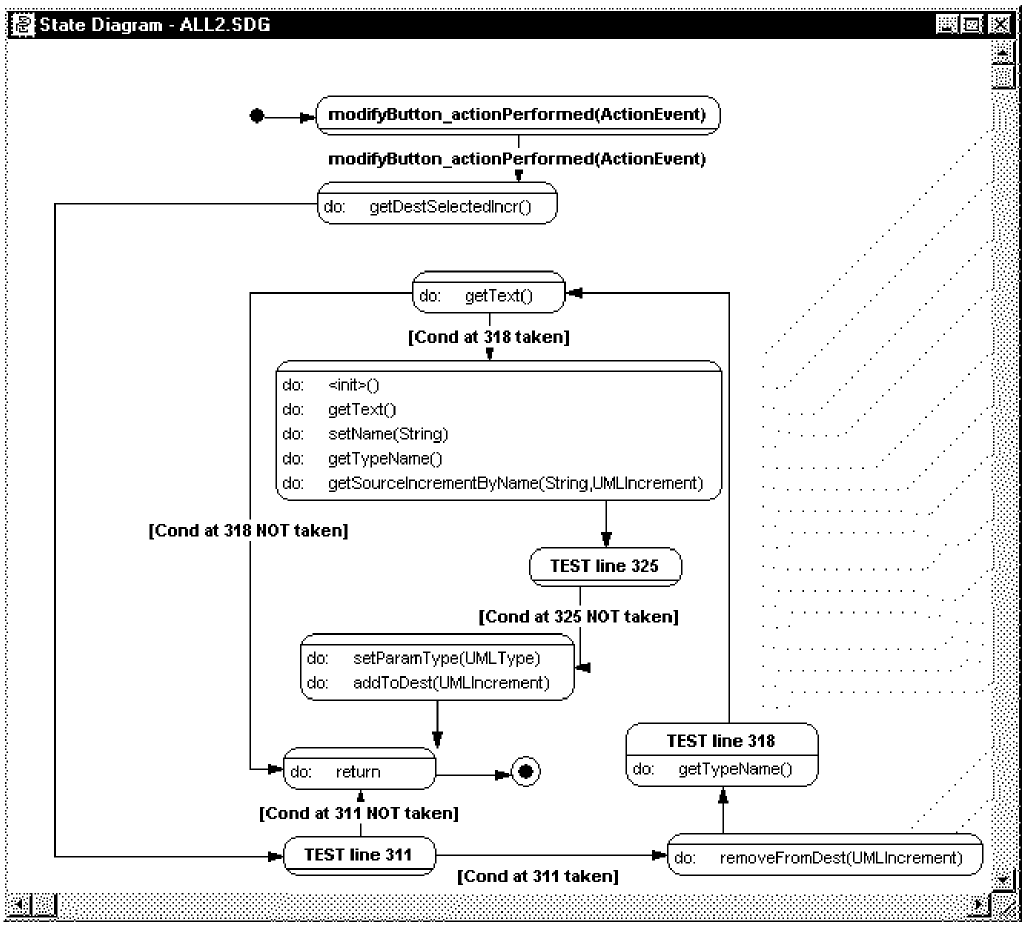
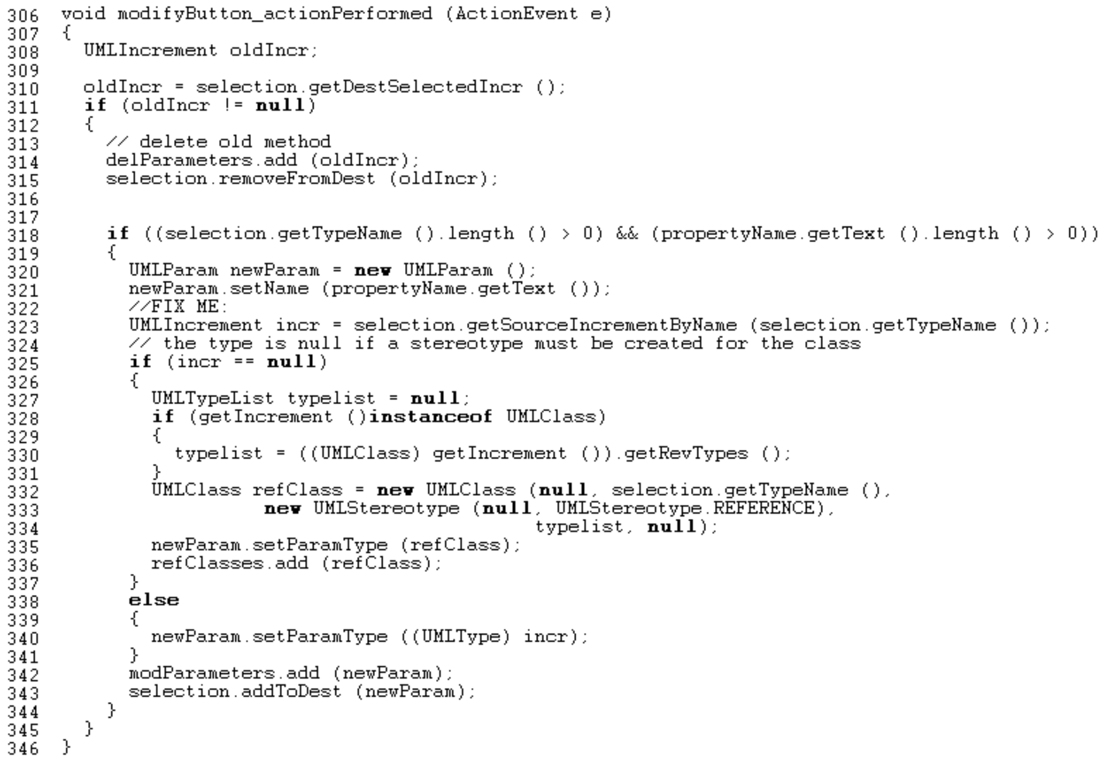


Figure 9.5: The detailed behavior of the method *modifyButton_actionPerformed (ActionEvent)* of class *de.uni_paderborn.fujaba.gui.PEParameters*. The state diagram describes the dynamic control flow of the method.



```
void modifyButton_actionPerformed (ActionEvent e)
{
  UMLIncrement oldIncr;

  oldIncr = selection.getDestSelectedIncr ();
  if (oldIncr != null)
  {
    // delete old method
    delParameters.add (oldIncr);
    selection.removeFromDest (oldIncr);


    if ((selection.getTypeName ().length () > 0) && (propertyName.getText ().length () > 0))
    {
      UMLParam newParam = new UMLParam ();
      newParam.setName (propertyName.getText ());
      //FIX ME:
      UMLIncrement incr = selection.getSourceIncrementByName (selection.getTypeName ());
      // the type is null if a stereotype must be created for the class
      if (incr == null)
      {
        UMLTypeList typelist = null;
        if (getIncrement ()instanceof UMLClass)
        {
          typelist = ((UMLClass) getIncrement ()).getRevTypes ();
        }
        UMLClass refClass = new UMLClass (null, selection.getTypeName (),
                  new UMLStereotype (null, UMLStereotype.REFERENCE),
                                        typelist, null);
        newParam.setParamType (refClass);
        refClasses.add (refClass);
      }
      else
      {
        newParam.setParamType ((UMLType) incr);
      }
      modParameters.add (newParam);
      selection.addToDest (newParam);
    }
  }
}
```

Figure 9.6: The source code of the method *modifyButton_actionPerformed(ActionEvent)* of class *de.uni_paderborn.fujaba.gui.PEParameters*

control flow and the internal behavior of the method. Dynamic control flows are useful for detecting decision making, for profiling, for studying the cyclomatic complexity, for investigating code coverage, etc. Since the dynamic control flow is constructed automatically based on the usage of the target software, the user can generate desired dynamic slices of the implicit control flow. This approach helps the user to get only those pieces of information she is interested in.

On the other hand, the state diagram can be used to estimate if the “test cases” (usage of the software) are covering enough. By comparing it to the source code or the static model, it can also be used to estimate if the method in question includes unreachable code. The points in the state diagram that reveal such pieces of information are the states with state names. A complete state diagram should show all possible branches. In other words, there should be (at least) two different paths leading out from each state that identifies a branching point. For example, the state diagram in Figure 9.5 is not complete, because from the state *TEST line 325* there is a single path to the final state. Note that this can not be concluded from the state *TEST line 318*, since the actual branching point is the next state, that is, a state with the action *getText()*.

9.3.2 Modeling the usage of a dialog

In this section we examine the overall behavior of the dialog introduced in Figure 9.2. Again, by running a few scripts in Rigi, all the nodes that belong to the dialog, nodes that depend on them, and nodes that are referenced by them can easily be separated from the rest of the system. Breakpoints were set at the first line of all chosen member functions and for all conditional statements.

The dialog was used three different times while running FUJABA (i.e., three instances of the class *de.uni_paderborn.fujaba.gui.PEParameters* were created). The objects were not alive at the same time. The usage of the dialog was the following:

1. The dialog was opened, a parameter name and type was defined, the *add* button was pressed, and the *ok* button was pressed.

2. The dialog was opened, the name of the parameter was changed, the *modify* button was pressed, and finally the *ok* button was pressed.

3. The dialog was opened, the *remove* button was pressed (a parameter was selected), and the *ok* button was pressed.

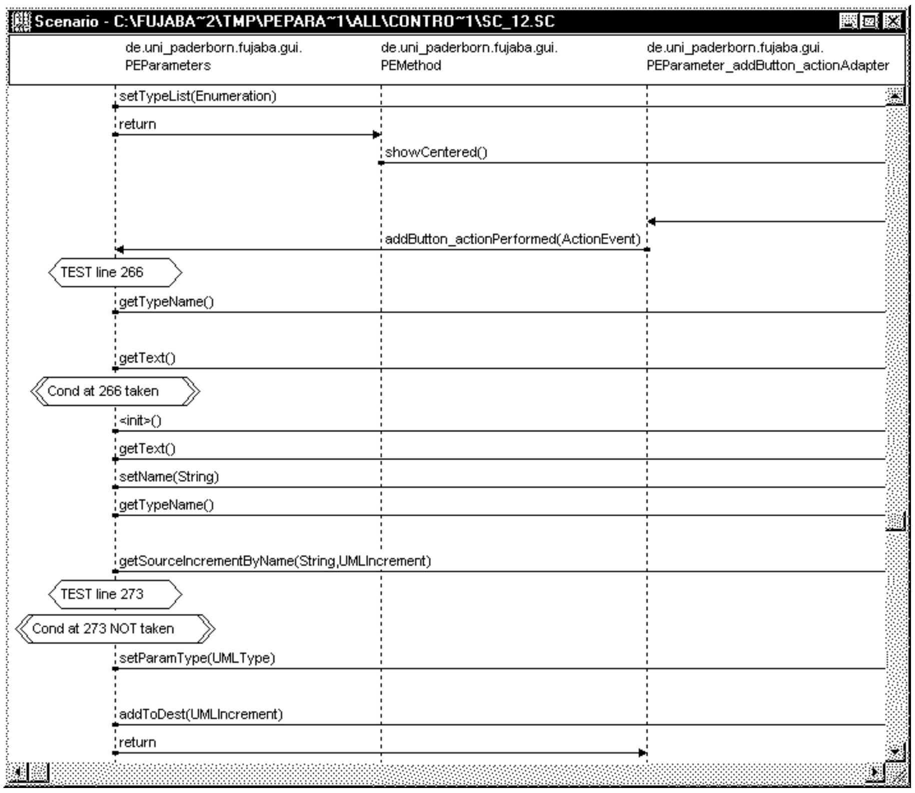

Scenario - C:\FUJABA~2\TMP\PEPARA~1\ALL\CONTRO~1\SC_12.SC
de.uni_paderborn.fujaba.gui.
PEParameters
de.uni_paderborn.fujaba.gui.
PEMethod
de.uni_paderborn.fujaba.gui.
PEParameter_addButton_actionAdapter
setTypeList(Enumeration)
return
showCentered()
addButton_actionPerformed(ActionEvent)
TEST line 266
getTypeName()
getText()
Cond at 266 taken
<init>()
getText()
setName(String)
getTypeName()
getSourceIncrementByName(String,UMLIncrement)
TEST line 273
Cond at 273 NOT taken
setParamType(UMLType)
addToDest(UMLIncrement)
return

Figure 9.7: An example scenario resulting when reverse engineering the behavior of the dialog in Figure 9.2.

Note that the above usage of the dialog does not cover all possible cases and hence cannot define the total behavior of the system. Altogether 20 scenarios result when using the dialog in the described way. Figure 9.7 shows a corner of one resulting scenario diagram, and Figure 9.8 describes the total behavior as a state diagram. As can be seen from Figure 9.8 the state diagrams tend to get rather big even as a result of a relatively simple and brief usage of the system. However, it can be assumed that the more the dialog is used, the less the synthesized state diagram will grow: rather than more states, more transitions would be generated describing different paths through the state diagram. When using the dialog the *modify* button was pressed once after changing the name of a parameter. In case of modeling the behavior of the button itself in Section 9.3.1, it was used four times in different situations. If that usage of the *modify* button were part of the usage when modeling the behavior of the whole dialog, then the state diagram in Figure 9.5 would be a sub-diagram of the state diagram generated for the class *de.uni_paderborn.fujaba.gui.PEParameters* itself. Now, only part of the state diagram in Figure 9.5 can be found from the state diagram in Figure 9.8. That part is located in the bottom right corner in the state diagram and is zoomed in Figure 9.9.

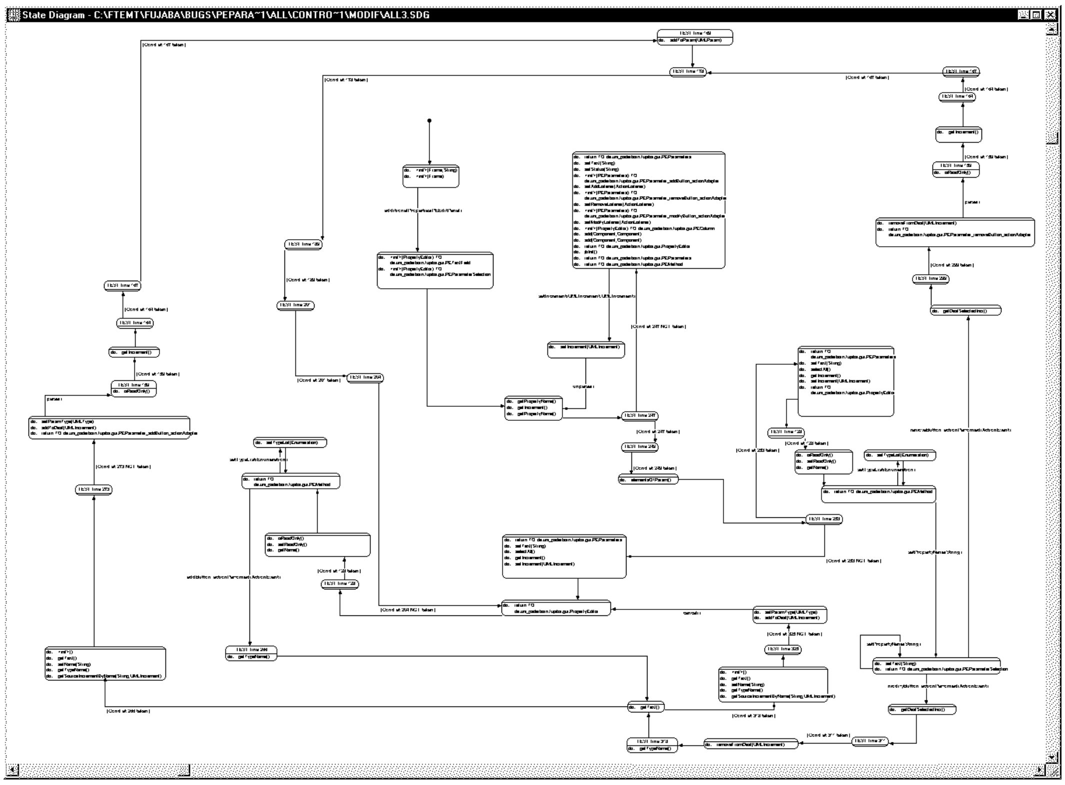

State Diagram - C:\FTEMT\FUJABA\BUGS\PEPARA~1\ALL\CONTRO~1\MODIF\ALL3.SDG

Figure 9.8: A state diagram showing the total behavior of the dialog in Figure 9.2. Both method invocations and dynamic control flow information were generated.

When breakpoints were only set for methods, constructors, and static blocks, information about branching inside methods was not generated. This results in a smaller and slightly more abstract state diagram as shown in Figure 9.10. The execution cases are the same as described above.

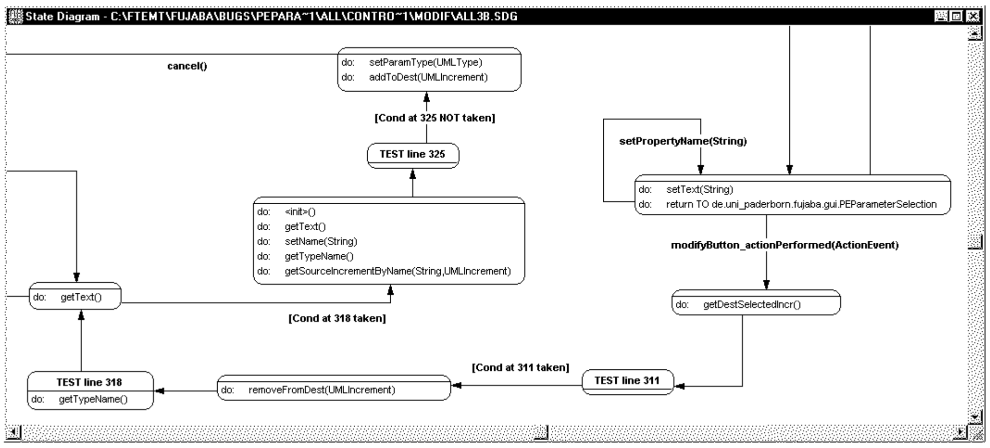

State Diagram - C:\FTEMT\FUJABA\BUGS\PEPARA~1\ALL\CONTRO~1\MODIF\ALL3B.SDG
cancel()
do: setParamType(UMLType)
do: addToDest(UMLIncrement)
[Cond at 325 NOT taken]
TEST line 325
setPropertyName(String)
do: setText(String)
do: return TO de.uni_paderborn.fujaba.gui.PEParameterSelection
do: <init>()
do: getText()
do: setName(String)
do: getTypeName()
do: getSourceIncrementByName(String,UMLIncrement)
modifyButton_actionPerformed(ActionEvent)
do: getText()
do: getDestSelectedIncr()
[Cond at 318 taken]
[Cond at 311 taken]
TEST line 318
do: getTypeName()
do: removeFromDest(UMLIncrement)
TEST line 311

Figure 9.9: The bottom right corner of the state diagram in Figure 9.8 depicts the influence of pressing the *modify* button.

A class is a basic software component in an object-oriented software system. Understanding its usage and the behavior of its instances is important (e.g., for reverse engineering, reuse, and forward engineering purposes). Such pieces of information would thus be valuable to be added to the documentation of the software. In the example of this section, several instances were created of the class *de.uni_paderborn.fujaba.gui.PEParameters*, but they were not alive concurrently. In Section 9.3.4 we study an example where this is not the case.

9.3.3 Structuring scenarios with behavioral patterns

The state diagram generation facility in SCED provides a powerful way to examine the total behavior of a class, object, or method disconnected from the rest of the system. Scenario diagrams, in turn, provide a view and an editor to browse the exact sequential event trace information. However, the amount of scenarios grows rapidly during the execution of the system. Hence, behavioral patterns are used for structuring, reorganizing, and packaging the scenario diagrams without changing their information contents. The patterns are searched using algorithms that apply the Boyer-Moore string matching algorithm [12]. Thus, the behavioral patterns are recognized based on strict string-based matches only. A pattern is a sequence of any SCED scenario diagram items repeated at least once.

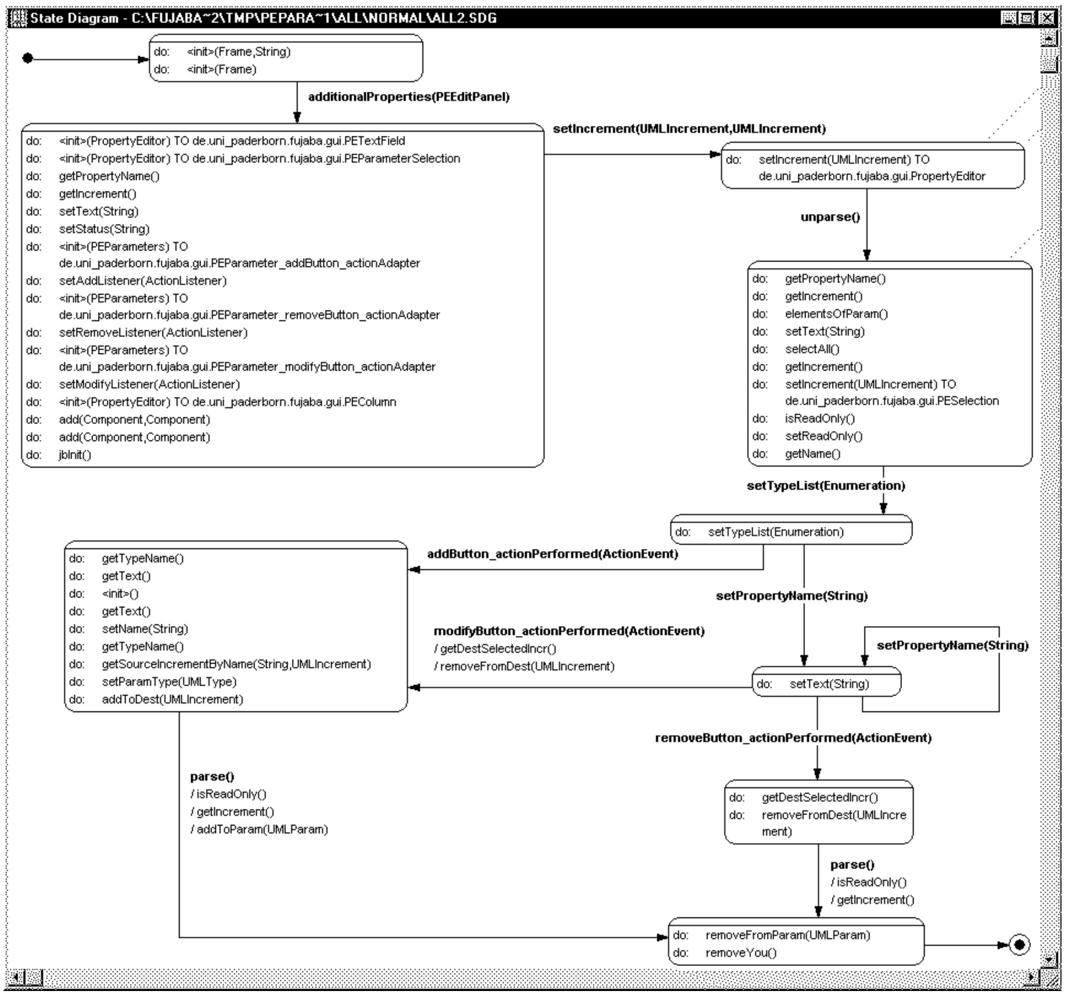

State Diagram - C:\FUJABA~2\TMP\PEPARA~1\ALL\NORMAL\ALL2.SDG
do: <init>(Frame,String)
do: <init>(Frame)
additionalProperties(PEEditPanel)
do: <init>(PropertyEditor) TO de.uni_paderborn.fujaba.gui.PETextField
do: <init>(PropertyEditor) TO de.uni_paderborn.fujaba.gui.PEParameterSelection
do: getPropertyName()
do: getIncrement()
do: setText(String)
do: setStatus(String)
do: <init>(PEParameters) TO de.uni_paderborn.fujaba.gui.PEParameter_addButton_actionAdapter
do: setAddListener(ActionListener)
do: <init>(PEParameters) TO de.uni_paderborn.fujaba.gui.PEParameter_removeButton_actionAdapter
do: setRemoveListener(ActionListener)
do: <init>(PEParameters) TO de.uni_paderborn.fujaba.gui.PEParameter_modifyButton_actionAdapter
do: setModifyListener(ActionListener)
do: <init>(PropertyEditor) TO de.uni_paderborn.fujaba.gui.PEColumn
do: add(Component,Component)
do: add(Component,Component)
do: jbInit()
setIncrement(UMLIncrement,UMLIncrement)
do: setIncrement(UMLIncrement) TO de.uni_paderborn.fujaba.gui.PropertyEditor
unparse()
do: getPropertyName()
do: getIncrement()
do: elementsOfParam()
do: setText(String)
do: selectAll()
do: getIncrement()
do: setIncrement(UMLIncrement) TO de.uni_paderborn.fujaba.gui.PESelection
do: isReadOnly()
do: setReadOnly()
do: getName()
setTypeList(Enumeration)
do: setTypeList(Enumeration)
addButton_actionPerformed(ActionEvent)
do: getTypeName()
do: getText()
do: <init>()
do: getText()
do: setName(String)
do: getTypeName()
do: getSourceIncrementByName(String,UMLIncrement)
do: setParamType(UMLType)
do: addToDest(UMLIncrement)
setPropertyName(String)
modifyButton_actionPerformed(ActionEvent)
/ getDestSelectedIncr()
/ removeFromDest(UMLIncrement)
setPropertyName(String)
do: setText(String)
removeButton_actionPerformed(ActionEvent)
parse()
/ isReadOnly()
/ getIncrement()
/ addToParam(UMLParam)
do: getDestSelectedIncr()
do: removeFromDest(UMLIncrement)
parse()
/ isReadOnly()
/ getIncrement()
do: removeFromParam(UMLParam)
do: removeYou()

Figure 9.10: A state diagram showing the member function invocations related to the dialog in Figure 9.2 (without internal branching in methods)

The application of the scenario structuring algorithms results in a hierarchical set of scenarios. The algorithms were applied for six scenarios that followed from a brief usage of FUJABA. During the FUJABA session, a dialog for defining and editing methods of a class was used a few times. The event trace information was generated for the dialog class *de.uni_paderborn.fujaba.gui.PEMethod* itself, for its superclasses *de.uni_paderborn.fujaba.*
gui.PETextEditor and *de.uni_paderborn.fujaba.gui.PropertyEditor*, and for class *de.uni_*
paderborn.fujaba.uml.UMLMethod. The superclasses were easily found using Rigi. Figure 9.11 shows one of the scenario diagrams after it has been modified by the scenario structuring algorithms. The scenario depicts the object interaction that was involved when the dialog was opened for a class that has four methods. The dialog was exited right after opening it by pressing the *cancel* button.

The object interaction that was needed for the initialization of the dialog was repeated every time the dialog was opened. The scenario structuring algorithm recognized that pattern and formed the subscenario box *subsc_7.sc* from that interaction. Figure 9.12 shows the contents of the subscenario box *subsc_7.sc*. The original scenarios consist of 473 scenario items. After applying the scenario structuring algorithm the total number of scenario items was reduced to 201, giving 58% savings.

The behavioral patterns were also searched from a larger set of scenarios. The algorithms were applied for the 20 scenario diagrams generated in the example case discussed in Section 9.3.2. The original scenarios consist of 1145 scenario items, which in this case are events, state boxes, assertion boxes, or repetition blocks. Note that even the original scenarios may contain repetition blocks if a single event is repeated multiple times in a row. There were 63 repetition blocks in those scenarios. After applying the scenario structuring algorithm the total number of scenario items was reduced to 484, giving 58 % savings. The scenario items appear in 39 different scenarios, 19 of them being newly created subscenarios. The 484 scenario items include 31 repetition blocks and 50 subscenario boxes. Note that the number of repetition boxes was actually decreased.

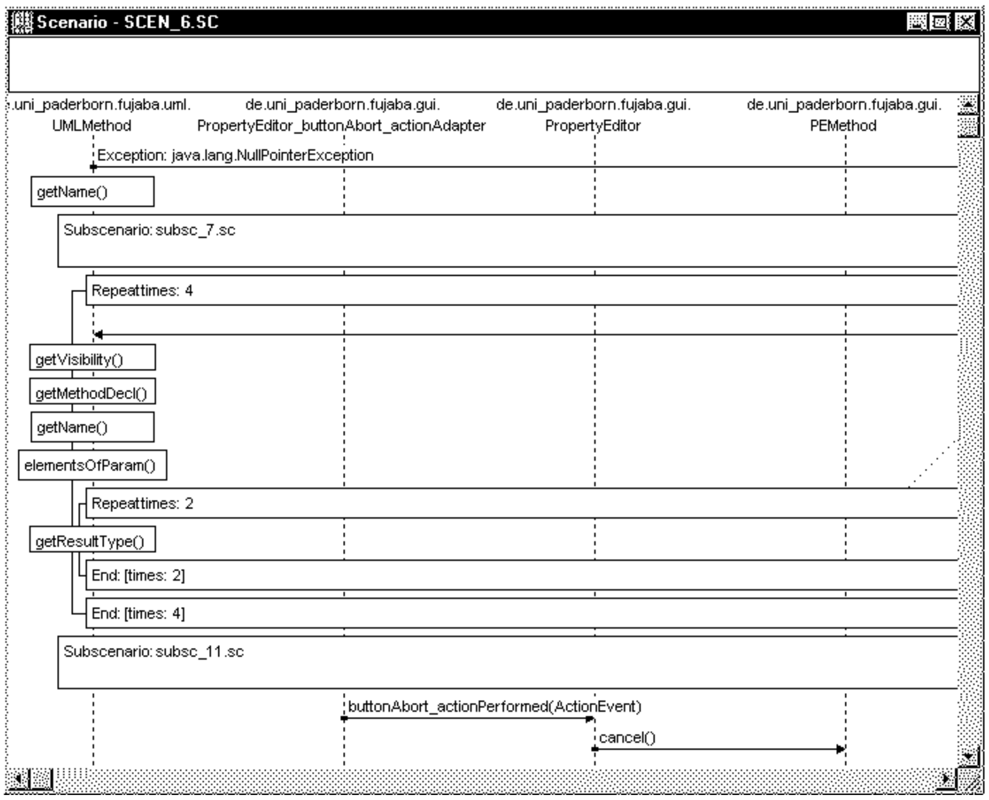


Figure 9.11: A corner of a scenario diagram that results when applying the scenario structuring algorithm. The algorithm has generated two subdiagram boxes and one repetition block with label *times: 4*. The other repetition block that contains a single scenario item was included in the original scenario.

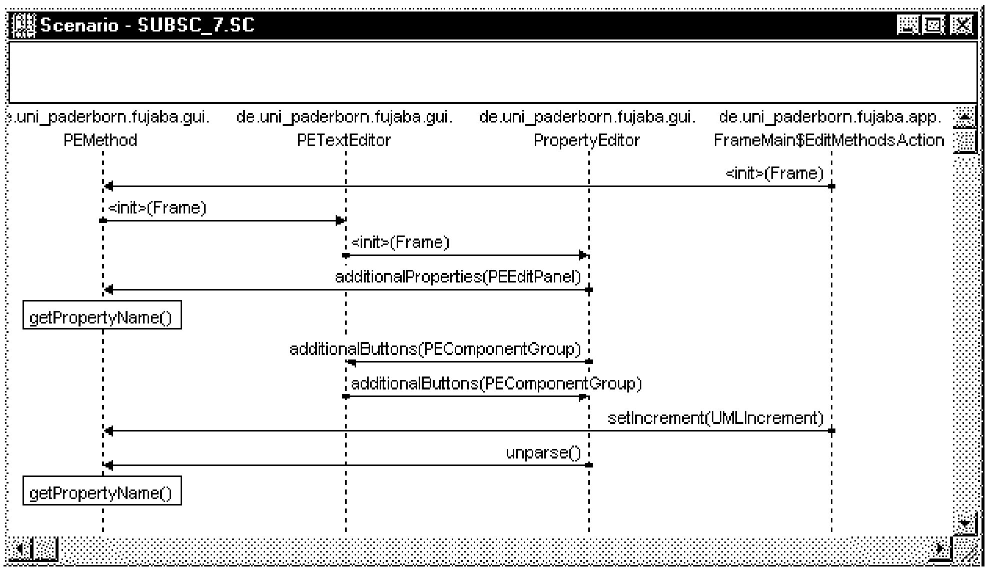

Scenario - SUBSC_7.SC
.uni_paderborn.fujaba.gui.
PEMethod
de.uni_paderborn.fujaba.gui.
PETextEditor
de.uni_paderborn.fujaba.gui.
PropertyEditor
de.uni_paderborn.fujaba.app.
FrameMain$EditMethodsAction
<init>(Frame)
<init>(Frame)
<init>(Frame)
additionalProperties(PEEditPanel)
getPropertyName()
additionalButtons(PEComponentGroup)
additionalButtons(PEComponentGroup)
setIncrement(UMLIncrement)
unparse()
getPropertyName()

Figure 9.12: The contents of the subdiagram box *subsc_7.sc* is opened in a separate scenario diagram window.

This is due to the generation of subscenarios: some of the repetition blocks appear in subscenarios and are hence referenced several times.

In our dynamic reverse engineering approach the event trace generated at run-time is split into several smaller scenarios. Unless guided by the user, this is done automatically. The used scenario structuring algorithms recognize only those patterns that are totally included in a single scenario. However, there might also be behavioral patterns that are partly included in two scenarios; because of the point of split, they exist at the end of one scenario and at the beginning of the next scenario. Such patterns remain unrecognized by our algorithms. More patterns would hence be found, if they were taken into account as well. That would require changing the original division of the event trace. Yet, that might be questionable, because the user may want to record a specific interval of the execution and save it in scenarios in a desired way. Another restriction is that our approach allows strict string-based matches only. In Ovation [27, 26] and ISVis [51], for example, the user has choices to define the rules for matches and hence to decide what is regarded as "similar behavior".

9.3.4 Modeling the behavior of a thread object

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

The behavior of a single thread is examined next by generating a state diagram for its *run()* method. The event trace information in this case was generated for objects (i.e., senders and receivers of method calls were set to be objects). When examining the behavior of the dialog in Figure 9.2 event trace information for classes was enough; each usage of a dialog represents the life-time of one object, and the dialog was used so that no two objects were alive at the same time. This is not the case with instances of subclasses of *AEngine*. Several instances of these subclasses can and usually are created during the execution. If at least two objects are alive at the same time, then they both execute their *run()* method concurrently. If the event trace information were generated for classes, the behavior of those objects would then be “merged” and information about the objects themselves would be lost. In scenario diagrams the effect would be the same, if two participants (representing objects) were collapsed together. For these reasons, the event trace information needs to be generated for objects.

The behavior of an instance of class *AInheritanceCheckerEngine* is studied next. As in Section 9.3.1, breakpoints were set to the *run()* method of the class *AEngine* and for all methods that have a call dependency with it. Altogether 39 scenarios were generated as a result of the following usage of FUJABA: the application was started, a project was loaded, the class diagram of the project was edited (a couple of methods were edited), the edited class diagram was saved, and FUJABA was exited. From the resulting scenarios it can be noticed that 17 objects of the *AInheritanceCheckerEngine* class were created during the usage. Figure 9.13 shows a state diagram generated for one of them.

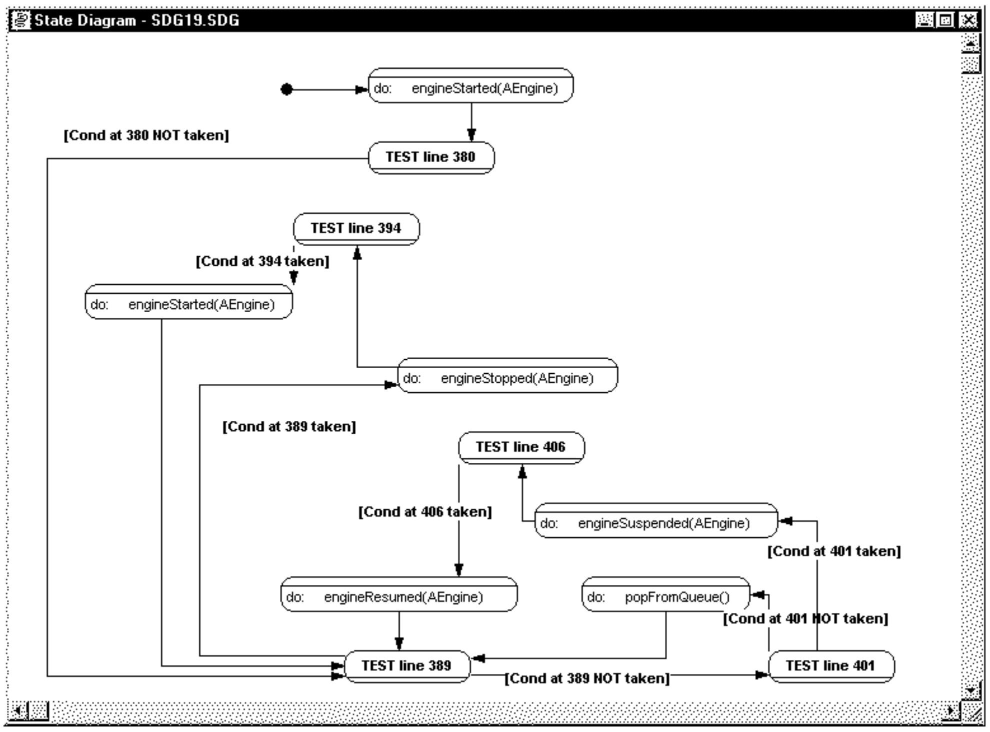


Figure 9.13: A state diagram for one instance of class *de.uni_paderborn.fujaba. analyze.AInheritanceCheckerEngine*

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. Hence, event trace information generated at the class level (i.e., senders and receivers of method calls are set to be classes) would not be enough for concluding the behavior of an instance of the *AInheritanceCheckerEngine* class. Scenario diagrams resulting when generating the event trace information at the class level can be seen as abstractions of the ones resulting when the information is generated at the object level. Such abstractions could also be built by modifying the scenario diagrams. A class level scenario diagram could be obtained from an object level scenario diagram by collapsing all participants representing instances of a single class to a new participant that represents the class itself.

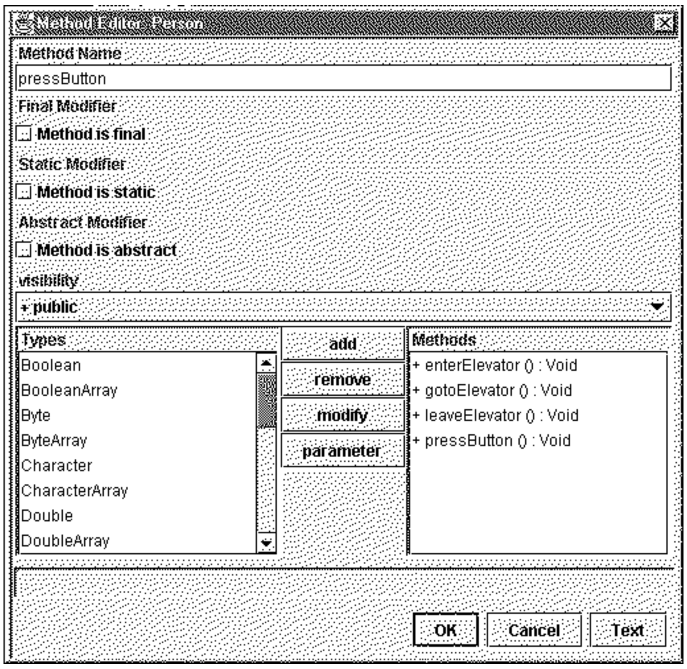
Objects and their interactions define the behavior of an object-oriented software system. Because of polymorphism, understanding the behavior is cumbersome. It is difficult, and in most cases impossible, to conclude the behavior of the objects by examining the source code. It was shown above that it might be impossible even if information about the run-time behavior were generated but the objects themselves were not identified. Means to describe the actual behavior of the objects are thus needed. The event trace information generated while running the target system tends to grow rapidly, especially, if the information is generated at the object level. One object may take part in tens of scenario diagrams. Browsing the scenarios can easily get difficult and troublesome. The state diagram synthesis approach provides a quick and efficient way to focus on the behavior of a single object.

9.3.5 Tracking down a bug

In FUJABA, methods can be created and edited using the dialog shown in Figure 9.14. 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 modify button, the parameters disappear. This bug was also mentioned in a FUJABA bug report [88]. The source of the bug was unknown.

The source of the bug was traced by examining scenarios generated when running FUJABA under JDebugger. In order to retrieve run-time information about the dialog in Figure 9.14, breakpoints were set for the dialog class *de.uni_paderborn.fujaba.gui.PEMethod* itself and for all its dependent classes. Information about the dynamic control flow was not generated. The usage of the dialog included pressing the *modify* button when a method with at least one parameter was selected.

By studying the resulting scenario diagrams it can be noticed 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 to a method. This



pressButton

+ public

Boolean
BooleanArray
Byte
ByteArray
Character
CharacterArray
Double
DoubleArray

+ enterElevator () : Void
+ gotoElevator () : Void
+ leaveElevator () : Void
+ pressButton () : Void

Figure 9.14: A dialog used for defining and editing methods of classes

can be concluded from the top of the scenario in Figure 9.15. The later part of the same scenario in Figure 9.16 shows that when the *modify* button is pressed (i.e., the method *modifyButton_actionPerformed(ActionEvent)* is called) a new instance of the class *UMLMethod* is created. This can be concluded from the constructor invocation *<init>()* of class *UMLMethod*. However, none of the scenarios generated after the one in Figures 9.15 and 9.16 have any events from the class *UMLMethod* to the class *UMLParam* directly or indirectly. If the old method is replaced by the new one (this can be assumed since a new method has been created), this might refer to the reason why the parameters were lost: the parameters from the old method are not copied to the new method created. This turned out to be the case.

The objects taking part in the execution, events sent and received by them, and the order of the events are important pieces of information for debugging a software system. The SCED scenario notation provides a descriptive and clear way to visualize them. For further analysis, the user can generate state diagrams for objects of interest. In the above case, for example, a state diagram could be generated for the class *UMLMethod* from the scenario in Figures 9.15 and 9.16 and all

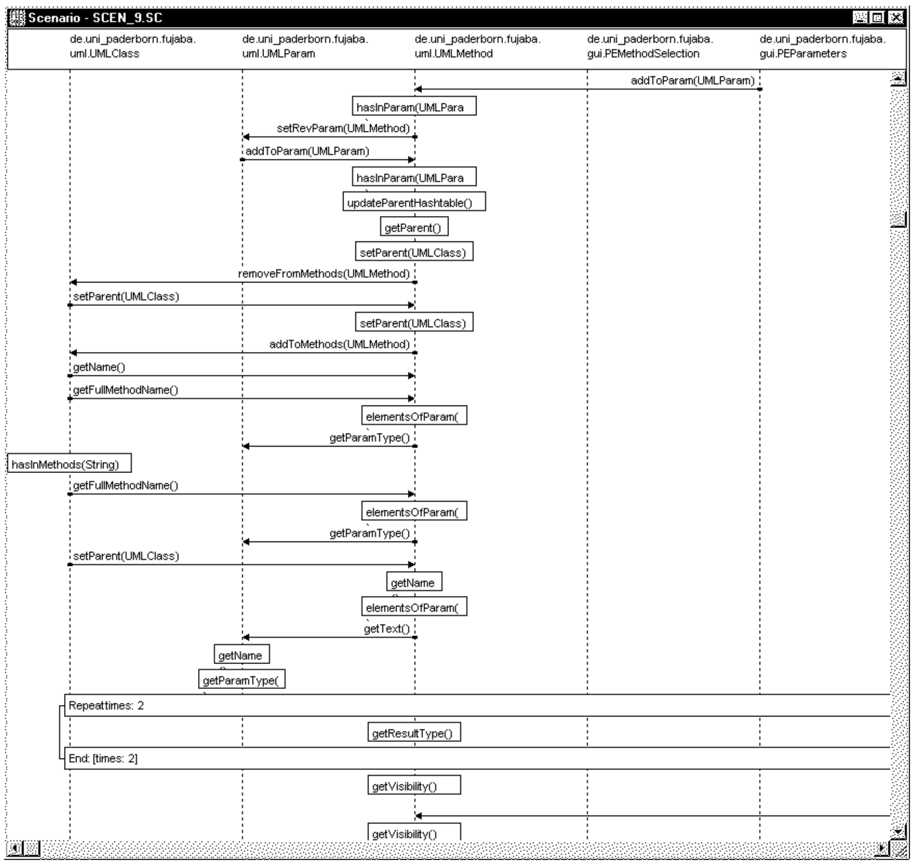


Figure 9.15: An example scenario diagram resulting when reverse engineering the behavior of the dialog in Figure 9.14

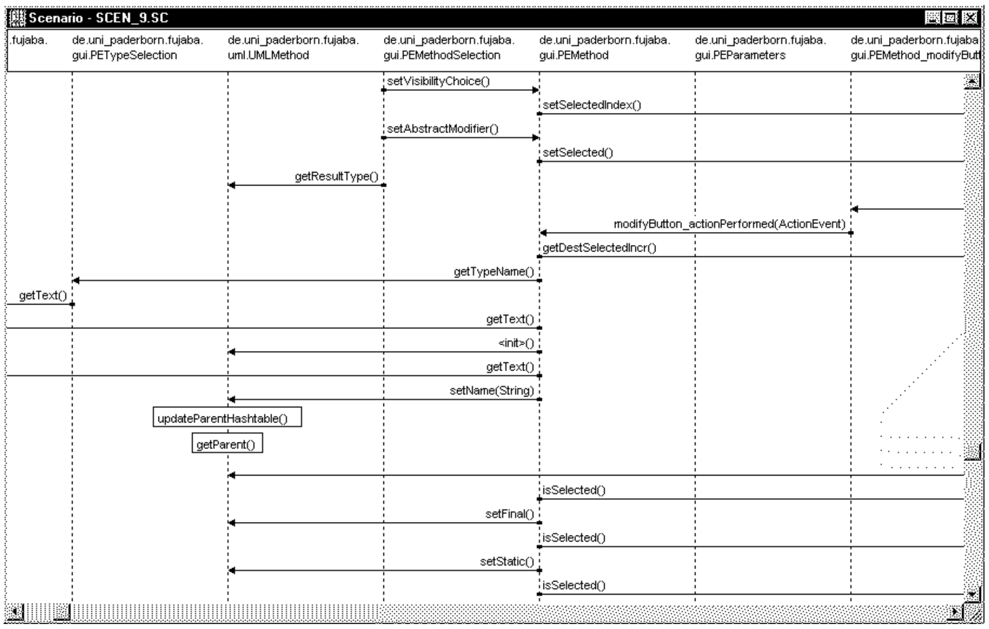


Figure 9.16: A later part of the scenario depicted in Figure 9.15.

scenarios generated after it. The resulting state diagram would show in one diagram the constructor invocation and the absence of the crucial method calls. Furthermore, another state diagram could be generated for the class *UMLMethod* after the bug has been fixed from scenarios generated under similar circumstances (same breakpoints and similar usage). By comparing the state diagrams the user could quickly notice the changed behavior. It would be useful to add such a diagram to the documentation.

9.4 Relationships between static and dynamic models

Static and dynamic views of the software usually share common information. For example, they both typically include information about software artifacts and their relations. The overlapping information forms a channel for information exchange among the views. Examples of such information exchange are presented in this section. The methods used in this section are described in Sections 8.7, 8.8, and 8.9.

9.4.1 Merging dynamic information into a static view

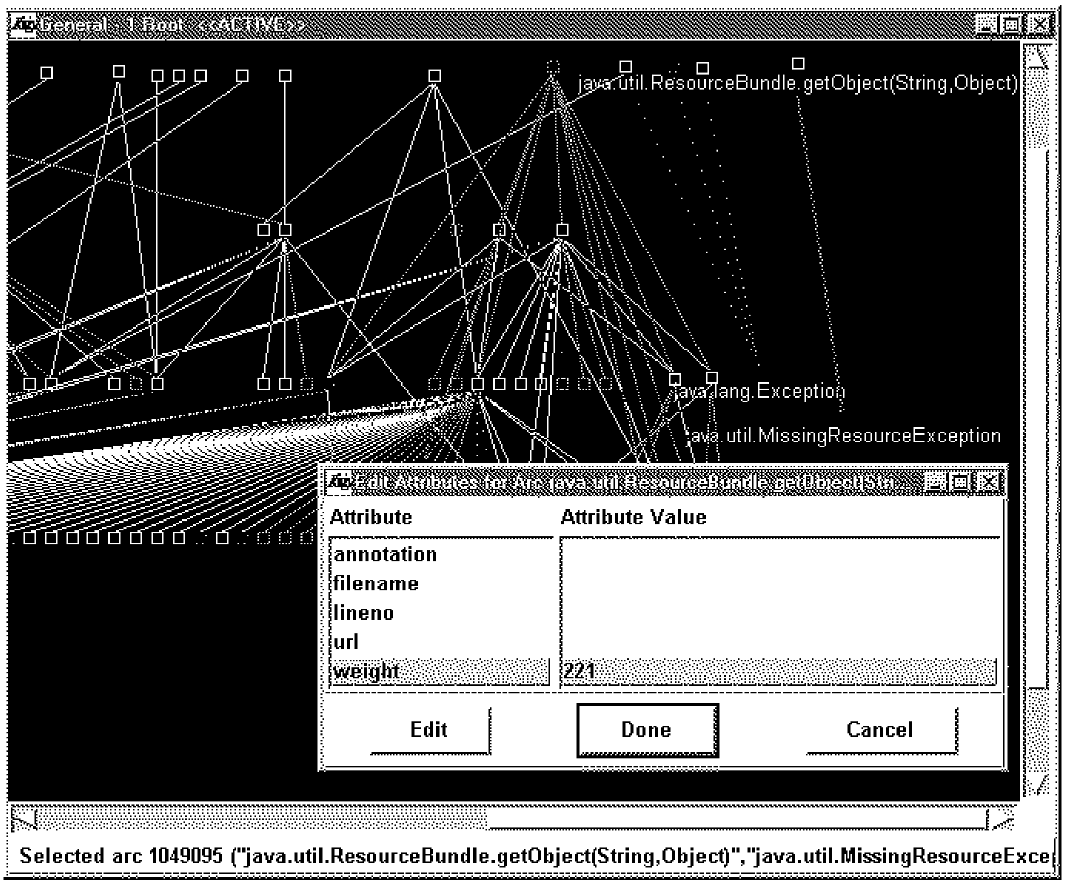
In Shimba, run-time information can be attached to a Rigi dependency graph. During the execution of the target system, weight values are given to *access*, *assign*, and *call* arcs in Rigi. The weight values indicate how many times these relationships have been used during the execution. Furthermore, *Exception* nodes and *throw* arcs are added according to their run-time usage. Figure 9.17 shows a Rigi graph to which run-time information has been added. To generate the dynamic information, breakpoints were set for all methods of the *de.uni_paderborn.fujaba. app.FujabaApp* and *de.uni_paderborn.fujaba.uml.UMLParam* classes. In addition, exception *java.util.MissingResourceException* was selected. FUJABA was used in the following way:

1. FUJABA was started,
2. a project was opened,
3. one parameter of a method of a class was removed,
4. the project was saved, and
5. FUJABA was exited.

In the Rigi view in Figure 9.17 a small dialog has been opened to show the attribute values of a *throw* arc entering the *java.util.MissingResourceException* node. The weight value of the arc indicates that the corresponding exception was thrown 221 times. The type of the exception and the large number of times it was thrown refers to an installation problem.

9.4.2 Slicing a Rigi view using SCED scenarios

The source of a known bug was traced in Section 9.3.5. To examine the structure of the parts of FUJABA that might contain the source of the bug, the initial Rigi graph in Figure 9.1, representing the whole FUJABA software, was sliced by a set of scenario diagrams that were generated after the dialog in Figure 9.14 was opened. There are altogether 29 such scenarios. The Rigi graph in Figure 9.18 results from applying Algorithm 4 to the initial Rigi graph. The node


General - 1 Root <<ACTIVE>>
java.util.ResourceBundle.getObject(String,Object)
java.lang.Exception
java.util.MissingResourceException
Edit Attributes for Arc java.util.ResourceBundle.getObject(Str...
Edit
Done
Cancel

Attribute	Attribute Value
annotation	
filename	
lineno	
url	
weight	221

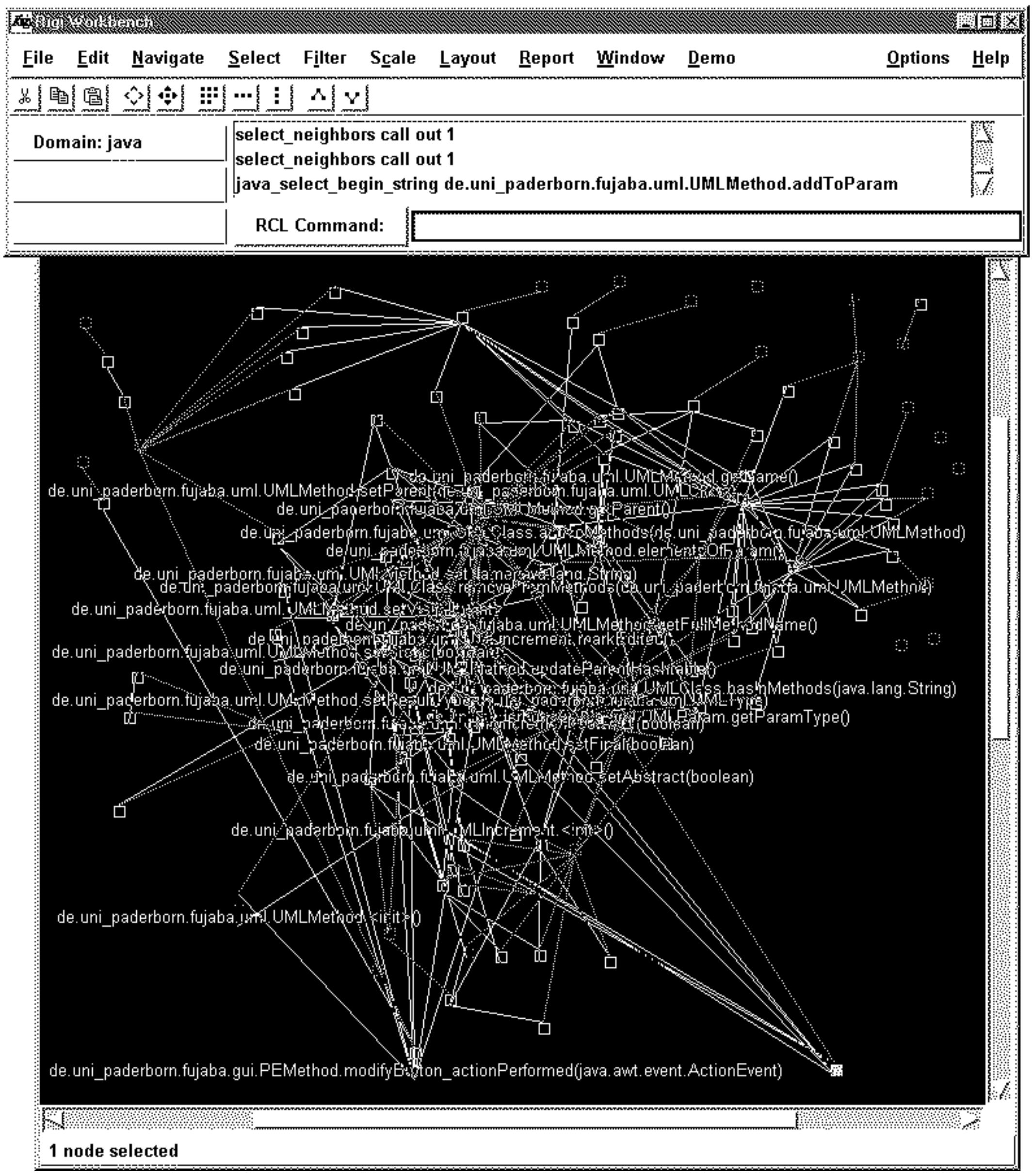
Selected arc 1049095 ("java.util.ResourceBundle.getObject(String,Object)","java.util.MissingResourceExce

Figure 9.17: A corner of a Rigi view containing both static and dynamic information. The user has selected a *throw* arc from node *java.util.Resource.getObject(String,Object)* to node *java.util.MissingResourceException* (the right most arc) and opened a dialog that shows the attribute values of that arc. The opened dialog shows that *java.util.MissingResourceException* has been thrown 221 times at run-time (the weight value of the selected arc).

modifyButton_actionPerformed(jawa.awt.event.ActionEvent) can be seen in the bottom left corner of the graph. The nodes, names of which have not been hidden, represent methods that can be called from it directly or indirectly. Such a chain of method calls, representing reachability of those methods from the *modifyButton_actionPerformed(java.awt.event.ActionEvent)* method, can be easily found by running a simple script that makes a forward reachability query. The default constructor of class *UMLMethod* belongs to this chain. This supports the conclusions of Section 9.3.5. By examining the scenarios we observed that when the *modify* button is pressed, a new method is created. We assumed that the new method is replaced with the old one. By running another script the node *addToParam(UMLParam)* can be found. It is placed in the bottom right corner of the graph. It is worth noting that it does not belong to the found chain of nodes. From this it can be concluded that the *addToParam(UMLParam)* method of class *UMLMethod* was not called (i.e., no parameters were added for methods) after the *modify* button was pressed. Moreover, *addToParam(UMLParam)* method of class *UMLMethod* cannot be called in any circumstances from the method *modifyButton_actionPerformed(java.awt.event.ActionEvent)* with the current implementation.

9.4.3 Raising the level of abstraction of SCED scenario diagrams using a high-level Rigi graph

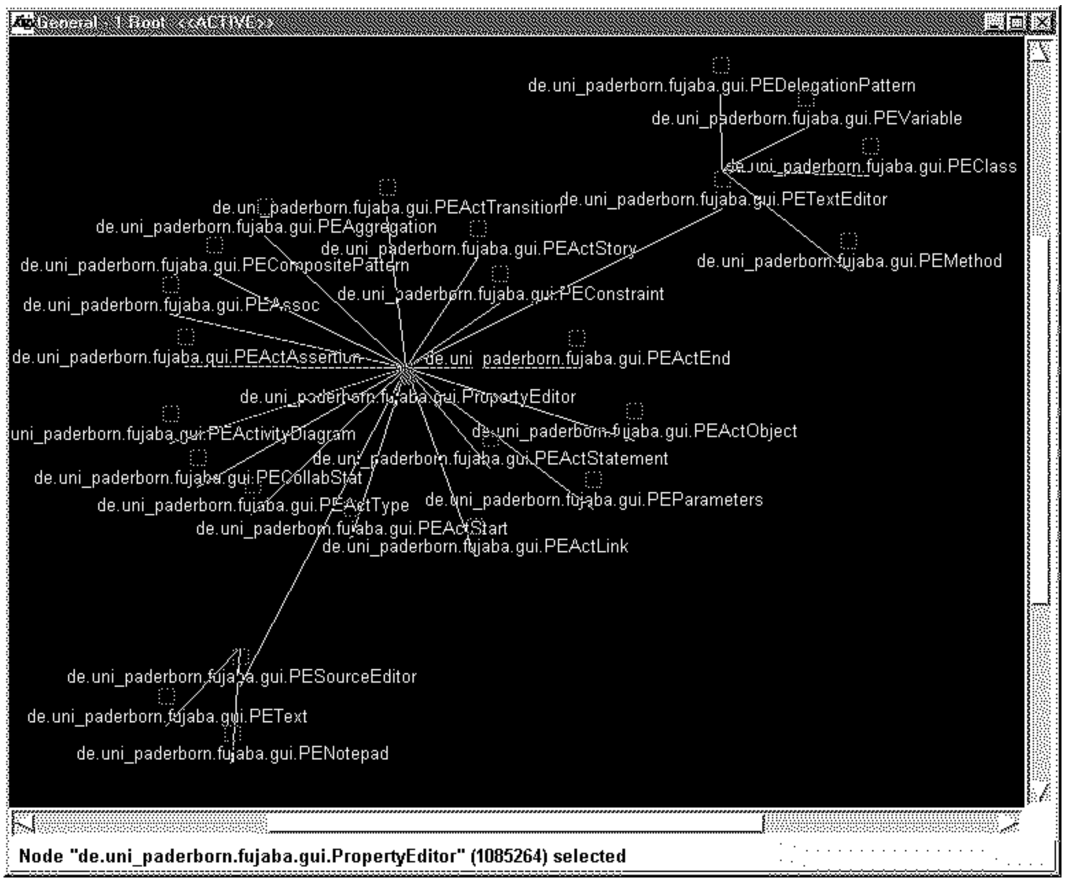
An example of modifying SCED scenarios using the approach described in Section 8.9 is discussed next. In FUJABA, classes that represent UML concepts belong to the package *de.uni_paderborn.fujaba.uml*. The editors and dialogs used for defining and editing these concepts are defined in the package *de.uni_paderborn.fujaba.gui*. For example, the dialog in Figure 9.14 is implemented as class *de.uni_paderborn.fujaba.gui.PEMethod*, and it is used for defining and editing methods that are represented as instances of class *de.uni_paderborn.fujaba.uml.UMLMethod*. Altogether 24 editor classes have the superclass *de.uni_paderborn.fujaba.gui.PropertyEditor*, from which the individual editors are derived. Figure 9.19 shows the class hierarchy of the editors. Similarly the class *de.uni_paderborn. fujaba.uml.UMLIncrement* is a super class for 46 classes that represent different UML notation concepts.


Rigi Workbench
File
Edit
Navigate
Select
Filter
Scale
Layout
Report
Window
Demo
Options
Help
Domain: java
RCL Command:
de.uni_paderborn.fujaba.uml.UMLMethod.setParent
de.uni_paderborn.fujaba.uml.UMLMethod.getParent()
de.uni_paderborn.fujaba.uml.UMLClass.addToMethods(de.uni_paderborn.fujaba.uml.UMLMethod)
de.uni_paderborn.fujaba.uml.UMLMethod.elementsOfParam()
de.uni_paderborn.fujaba.uml.UMLClass.removeFromMethods(de.uni_paderborn.fujaba.uml.UMLMethod)
de.uni_paderborn.fujaba.uml.UMLMethod.setStatic(boolean)
de.uni_paderborn.fujaba.uml.UMLMethod.setFullMethodName()
de.uni_paderborn.fujaba.uml.UMLIncrement.markEdited()
de.uni_paderborn.fujaba.uml.UMLMethod.updateParentHashtable()
de.uni_paderborn.fujaba.uml.UMLClass.hasInMethods(java.lang.String)
de.uni_paderborn.fujaba.uml.UMLMethod.setResultType(de.uni_paderborn.fujaba.uml.UMLType)
de.uni_paderborn.fujaba.uml.UMLParam.getParamType()
de.uni_paderborn.fujaba.uml.UMLMethod.setFinal(boolean)
de.uni_paderborn.fujaba.uml.UMLMethod.setAbstract(boolean)
de.uni_paderborn.fujaba.uml.UMLIncrement.<init>()
de.uni_paderborn.fujaba.uml.UMLMethod.<init>()
de.uni_paderborn.fujaba.gui.PEMethod.modifyButton_actionPerformed(java.awt.event.ActionEvent)

select_neighbors call out 1
select_neighbors call out 1
java_select_begin_string de.uni_paderborn.fujaba.uml.UMLMethod.addToParam

1 node selected

Figure 9.18: The graph resulting when the initial Rigi graph for FUJABA software in Figure 9.1 has been sliced by a set of scenarios


General : 1 Root <<ACTIVE>>
de.uni_paderborn.fujaba.gui.PEDelegationPattern
de.uni_paderborn.fujaba.gui.PEVariable
de.uni_paderborn.fujaba.gui.PEClass
de.uni_paderborn.fujaba.gui.PETextEditor
de.uni_paderborn.fujaba.gui.PEActTransition
de.uni_paderborn.fujaba.gui.PEAggregation
de.uni_paderborn.fujaba.gui.PEActStory
de.uni_paderborn.fujaba.gui.PEMethod
de.uni_paderborn.fujaba.gui.PECompositePattern
de.uni_paderborn.fujaba.gui.PEConstraint
de.uni_paderborn.fujaba.gui.PEAssoc
de.uni_paderborn.fujaba.gui.PEActAssertion
de.uni_paderborn.fujaba.gui.PEActEnd
de.uni_paderborn.fujaba.gui.PropertyEditor
uni_paderborn.fujaba.gui.PEActivityDiagram
de.uni_paderborn.fujaba.gui.PEActObject
de.uni_paderborn.fujaba.gui.PEActStatement
de.uni_paderborn.fujaba.gui.PECollabStat
de.uni_paderborn.fujaba.gui.PEParameters
de.uni_paderborn.fujaba.gui.PEActType
de.uni_paderborn.fujaba.gui.PEActStart
de.uni_paderborn.fujaba.gui.PEActLink
de.uni_paderborn.fujaba.gui.PESourceEditor
de.uni_paderborn.fujaba.gui.PEText
de.uni_paderborn.fujaba.gui.PENotepad

Node "de.uni_paderborn.fujaba.gui.PropertyEditor" (1085264) selected

Figure 9.19: A Rigi view showing the class *de.uni_paderborn.gui.PropertyEditor* and all its subclasses

For capturing high-level information about the Model-View structure of the FUJABA class diagram editor static abstractions were made. First, class *de.uni_paderborn.gui.PropertyEditor* and all its subclasses were collapsed into a single *de.uni_paderborn.gui.PropertyEditor2* node. Second, class *de.uni_paderborn.uml.UMLIncrement* and all its subclasses were collapsed into a single *de.uni_paderborn.uml.UMLIncrement2* node. Third, all classes that implement the *java.awt.event.ActionListener* interface were collapsed into a single *java.awt.event. ActionListener2* node. There were altogether 92 such classes. Those classes represent GUI items that are able to receive action events (e.g., user inputs). Hence, they represent the system border. Fourth, class *de.uni_paderborn.gui.PESelection* and all its subclasses were collapsed into a single *de.uni_paderborn.gui.PESelection2* node. There are only six such subclasses. Class *de.uni_paderborn.gui.PESelection* is used for providing the user a list to which she can add items using the *add* button and remove items using the *remove* button.

A set of SCED scenario diagrams were generated when reverse engineering the dynamic interaction between model and view components of the FUJABA class diagram editor. Breakpoints were set for all methods of the following classes: *UMLMethod*, *UMLParam*, *UMLClass*, *UMLAttr*, *PEMethod*, *PEParameter*, *PEClass*, and *PEVariable*. The first four classes belong to the *de.uni_paderborn.fujaba.uml* package, and the rest of them belong to the *de.uni_paderborn.fujaba.gui* package.

The usage included adding and editing methods, variables, and parameters. In addition, a new class was added to the class diagram. After applying the method of raising the level of abstraction of SCED scenario diagrams using a high-level Rigi graph the number of scenarios was decreased from 26 to 7 and the number of scenario items was decreased from 2038 to 513. Figure 9.20 shows a corner of one of the resulting scenarios. The scenario shows a usage in which the *parameter* button of the dialog in Figure 9.14 (the dialog for editing methods) was pressed. That caused the dialog in Figure 9.2 (the dialog for editing parameters) to be opened. Using that dialog a new parameter was added to the method, and finally, the parameter was modified. The usage is clearly recognizable from the scenario. Furthermore, participants representing the system border (*javax.swing.AbstractButton* and *java.awt.event.ActionListener2*), the view components (*de.uni_paderborn.gui.PropertyEditor2* and *de.uni_paderborn.gui.PESelection2*), and the model components (*de.uni_paderborn.uml.UMLIncrement2*) can be easily distinguished. Note, that the system border always communicates with the model components through the view components. The high-level scenario thus clarifies the user interaction as well as the communication between the view and the corresponding model components.

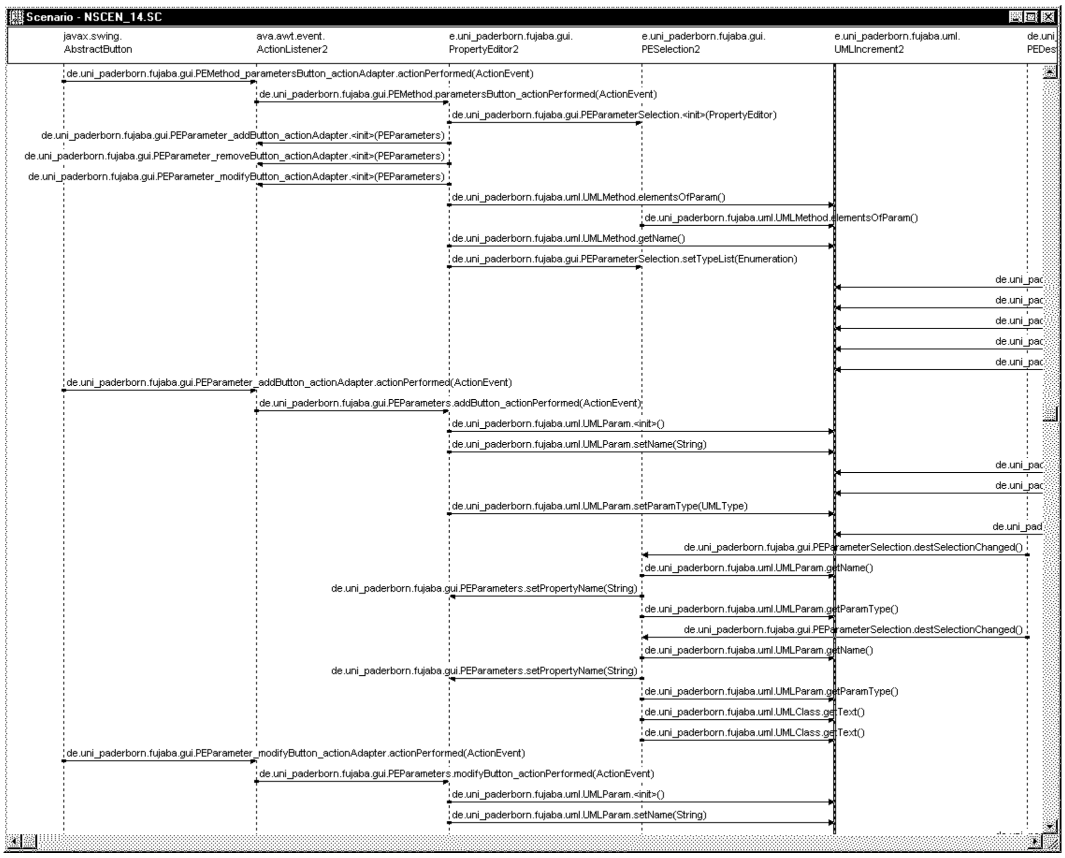

Scenario - NSCEN_14.SC
javax.swing.AbstractButton
ava.awt.event.ActionListener2
e.uni_paderborn.fujaba.gui.PropertyEditor2
e.uni_paderborn.fujaba.gui.PESelection2
e.uni_paderborn.fujaba.uml.UMLIncrement2
de.uni PEDes
de.uni_paderborn.fujaba.gui.PEMethod_parametersButton_actionAdapter.actionPerformed(ActionEvent)
de.uni_paderborn.fujaba.gui.PEMethod.parametersButton_actionPerformed(ActionEvent)
de.uni_paderborn.fujaba.gui.PEParameterSelection.<init>(PropertyEditor)
de.uni_paderborn.fujaba.gui.PEParameter_addButton_actionAdapter.<init>(PEParameters)
de.uni_paderborn.fujaba.gui.PEParameter_removeButton_actionAdapter.<init>(PEParameters)
de.uni_paderborn.fujaba.gui.PEParameter_modifyButton_actionAdapter.<init>(PEParameters)
de.uni_paderborn.fujaba.uml.UMLMethod.elementsOfParam()
de.uni_paderborn.fujaba.uml.UMLMethod.elementsOfParam()
de.uni_paderborn.fujaba.uml.UMLMethod.getName()
de.uni_paderborn.fujaba.gui.PEParameterSelection.setTypeList(Enumeration)
de.uni_pad
de.uni_paderborn.fujaba.gui.PEParameter_addButton_actionAdapter.actionPerformed(ActionEvent)
de.uni_paderborn.fujaba.gui.PEParameters.addButton_actionPerformed(ActionEvent)
de.uni_paderborn.fujaba.uml.UMLParam.<init>()
de.uni_paderborn.fujaba.uml.UMLParam.setName(String)
de.uni_paderborn.fujaba.uml.UMLParam.setParamType(UMLType)
de.uni_paderborn.fujaba.gui.PEParameterSelection.destSelectionChanged()
de.uni_paderborn.fujaba.uml.UMLParam.getName()
de.uni_paderborn.fujaba.gui.PEParameters.setPropertyName(String)
de.uni_paderborn.fujaba.uml.UMLParam.getParamType()
de.uni_paderborn.fujaba.uml.UMLClass.getText()
de.uni_paderborn.fujaba.gui.PEParameter_modifyButton_actionAdapter.actionPerformed(ActionEvent)
de.uni_paderborn.fujaba.gui.PEParameters.modifyButton_actionPerformed(ActionEvent)

Figure 9.20: A high-level SCED scenario diagram showing the interaction between high-level static components that have been composed using Rigi.

9.5 Discussion

Several tasks were set for the case study. In what follows we discuss how well these tasks were achieved, what problems and limitations of Shimba were encountered during the case study, and how useful the reverse engineering techniques of Shimba were in practice.

9.5.1 Results of the case study

All the tasks for the case study (cf. Section 9.1) were achieved. In some cases, the generated models gave also additional information that was not even expected. For example, when merging dynamic information into a static view an unexpected installation problem was encountered. However, the results were not equally good or useful in each case. Next we discuss two cases in which the results were not fully satisfying.

Searching for the behavioral patterns and structuring the SCED scenario 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 scenario diagrams with them, the readability of the original scenario diagrams got worse in some cases. 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 thus needs to be done manually. Furthermore, a pattern contains an arbitrary sequence of SCED scenario diagram items, formed on the basis of the length of the pattern. Thus, one subscenario might contain SCED scenario diagram items 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. The patterns that are repeated in succession are visualized as repetition blocks that are named by the number of iterations. Hence, such patterns do not decrease the readability of the scenario diagrams.

Modeling the overall behavior of a complicated object might result in a large state diagram that does not fit entirely in the SCED window in a readable form. The state diagram optimization algorithms are able to decrease the size of the state diagram dramatically. However, some of the synthesized state diagrams were too large even after the optimization. SCED allows the user to scroll the state diagram window, which enables the user to read its content. The window can also be zoomed out to decrease its size. If the state diagram is shrank to fit on the screen, the font in the state diagram might get unreadable. Section 9.3.2 shows an example of such a case.

9.5.2 Limitations of Shimba

The string matching algorithms that are used in Shimba for recognizing behavioral patterns from the SCED scenario diagrams have a few limitations. First, the algorithms are able to find patterns only if they lay entirely inside one SCED scenario diagram. Hence, the patterns that begin from the end of one scenario and continue in the next scenario are not recognized. Second, the proposed method is able to find exact matches only. It might be useful to give the user some freedom in defining what is considered to be similar behavior. Third, the string matching algorithms often produce several subscenarios. Again, an option to influence the construction of subscenarios would be desirable.

The state diagram synthesis algorithm is fast if control flow information is included in the SCED scenario diagrams. If that is not the case, the algorithm might need to backtrack several times, which makes the synthesis slow (the synthesis may take a few tens of seconds to complete).

The static information can be extracted from the Java byte code very efficiently and fast using the *JExtractor* tool. The dynamic information is extracted when running the target application under the *JDebugger* debugger. Because of the used technique, the execution of the target Java software system gets slow, especially when information for many objects is extracted and the control flow information is included.

Dynamic control flow information can be included in SCED scenario diagrams during the dynamic reverse engineering process. A hit of a conditional statement (e.g., “**if**(x)”) and an evaluation of its condition are visualized as a state box and an assertion box in a SCED scenario diagram, respectively. The name of the state box (e.g., *TEST line 24*) and the name of the assertion box (e.g., *Cond at 24 taken*) both refer to the line number of the conditional statement in the source code. Such names are not very informative nor useful. Since Shimba is independent of the source code of the target software system, the state boxes are not labeled by the actual conditional statements, even though that would be more descriptive. Another approach could be to give the assertion boxes names that refer to line numbers of statements that are executed right after the evaluation of the conditions.

9.5.3 Experiences with Shimba

Shimba supports both static and dynamic reverse engineering. The case study showed that it is useful to make the static analysis before the 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). Moreover, using Rigi scripts the engineer can quickly and easily find the part of interest and the neighboring parts. When the neighborhood is also selected, the engineer can be confident that the event trace contains all the information concerning the part of interest. Finally, analyzing the SCED scenario diagrams is easy in this case, because uninteresting information is not shown.

The state diagram synthesis facility of SCED, combined with the state diagram optimization technique, was very useful and practical for analyzing the total behavior of selected objects and methods, even when the resulting state diagram was large. To the best of our knowledge, such features are not provided by other dynamic reverse engineering tools. The state 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. That would be very difficult by browsing the scenario diagrams (or other kinds of MSCs). The state diagram optimization algorithms were used most of the times when state diagrams were generated, since they reduce the size of the state diagram significantly, make the diagram more readable, and emphasize similar behavior.

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 scenario 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 scenario 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 scenario diagrams that visualize the exceptional behavior. Such a graphical support for debugging helps the engineer to track down the bug and to conclude the reason quickly. The 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 scenarios are related to the rest of the system, and what is the underlying structure that causes this behavior. The 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 (i.e., information about the order in which things has happened). This piece of information is not included in the annotated Rigi graph.

In Shimba, the level of abstraction of the SCED scenario diagrams can be raised using static abstractions that are 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 scenario 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. The current implementation of this technique has its also down sides, the biggest one being the way the method calls in the resulting scenario diagram are written. Currently, the name of the package and the name of the class are inserted into each method call. This makes the scenario diagrams slightly hard to read. The method names are expanded this way, since the participants that represent high-level static components do not represent the actual senders and receivers of the method calls.

In practice, combining static and dynamic reverse engineering was very useful. All the techniques 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 useful for forward engineering as well. When constructing software systems, debugging facilities are needed. The engineer can also 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 SCED scenario and state diagrams and checked against use case specifications.

Chapter 10

Conclusions

10.1 Discussion

Chikofsky and Cross define reverse engineering as a process of analyzing a subject system to identify the system's components and their inter-relationships, and to create representations of the system in another form at higher levels of abstraction [18]. The former goal can be achieved by using information extraction tools and the latter goal by producing design models from the target software. In this research, both static and dynamic reverse engineering are considered, the emphasis being on dynamic reverse engineering. Static reverse engineering aims at modeling the static structure of the target software, while dynamic reverse engineering models its run-time behavior.

10.1.1 Modeling the target software

Most currently available reverse engineering tools focus on either static or dynamic aspects of the software but rarely on both. In forward engineering OOAD methodologies provide various models that can be used in analysis and design phases to model the static structure and the 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.

In this dissertation, the reverse engineering of Java software is discussed. Both static and dynamic models are built to help the user to analyze different aspects of the target software. Static information is extracted from Java class files and viewed using the Rigi reverse engineering environment [74]. The dynamic event trace information is generated by running the target software under a customized jdk debugger called *JDebugger*. This information is viewed as scenario diagrams using the SCED dynamic modeling tool [56]. 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. Moreover, string matching algorithms are used to structure the scenario diagrams generated and to raise their level of abstraction.

Both static and dynamic analysis contain information about software artifacts and their relations. The shared information enables information exchange among the models. The models are used to complement each other and to provide extended support for slicing the models. Chapter 8 discusses such features implemented in Shimba. Section 8.6 discusses how dynamic information can be merged into a static Rigi view. Method calls, for instance, can be given weight values, indicating their actual usage at run-time. In addition, new nodes are typically added to the view, for example, nodes representing exceptions thrown at run-time. The information hiding and filtering mechanisms of Rigi can then be used to view the software artifacts and relations that are used (heavily enough). Section 8.7 describes how a static Rigi graph is used to define the software artifacts for which dynamic information is generated. This approach is especially useful when the user is interested in the behavior of a certain part of a software system. It dramatically decreases the amount of event trace information generated, yet containing the information of interest. In Section 8.8, a method for slicing a Rigi graph by a set of SCED scenarios is discussed. The dynamic slicing approach presented can be used for finding structural flaws in the software that cause unexpected or undesired behavior. Section 8.9, in turn, discusses a method for compressing SCED scenarios based on the information included in a Rigi graph. In the proposed method the level of abstraction of SCED scenarios is raised using a high-level Rigi graph. The modified scenario shows interaction among high-level static components. This approach helps the user to understand the overall behavior of the software.

In this research we have not combined the state diagram synthesis facility of SCED with the mechanism for raising the level of abstraction of SCED scenarios using a high-level Rigi graph. However, there are obvious possibilities to support understanding the behavior of high-level static components by combining these two techniques. Furthermore, as in the approach of slicing the Rigi graph with SCED scenario diagrams, discussed in Section 8.8, the static Rigi graph could be used for slicing SCED scenarios. The results would be similar to using the Rigi graph to guide the generation of dynamic information (cf. Section 8.7). In some cases, however, the user might not know which objects are the interesting ones before generating the run-time information. The user can, naturally, delete events and participants from the scenario diagram by using the SCED scenario diagram editor. However, a more efficient and profound slicing mechanism would be achieved, if the static dependency graph could be used to define the parts of the software (e.g., related according to some criteria) to be filtered out. Such an approach would help the user to understand and browse the event trace information.

10.1.2 Applying reverse engineering approaches to forward engineering

The application of reverse engineering techniques is not limited to understanding old legacy systems. They can and should be applied to support forward engineering as well. In software development reverse engineering the current static structure of the software helps the engineer to ensure 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, debugging, understanding and ensuring the current behavior of the software system, etc. Applying reverse engineering techniques during the software development phase also supports documentation, hence avoiding ending up in the similar situation with Java code, as we currently face with legacy COBOL and C code.

Ideally, reverse engineering tools applied to object-oriented software systems should be able to produce standard OOAD models from the target software. Since such models are familiar to the user, this would unburden her from learning yet another model or diagram notation. Moreover, if the models used in forward and reverse engineering are the same, the tools would be able to give more support for re-engineering and round-trip-engineering.

In Shimba, SCED scenario diagrams are used for modeling object interactions. In UML [95, 85] both sequence diagrams and collaboration diagrams are used for modeling object interactions. Collaboration diagrams do not show time as a separate dimension, as sequence diagrams do. However, they show relationships among the objects explicitly, hence including both dynamic and static aspects of the software. Shimba could be used to extract both the static information and the event trace information needed for constructing collaboration diagrams. The usage of collaboration diagrams should complement but not replace the usage of scenario diagrams. Scenario diagrams provide a simple and flexible way to show object interaction: SCED scenarios may contain an arbitrary event sequence among an arbitrary set of participants (cf. Section 4.1.1). The generated event trace information typically contains a large set of objects and executed operations. One collaboration diagram, in turn, usually shows an execution of one operation or use case. Constructing and connecting collaboration diagrams so that the whole event trace could be expressed with them is not a simple task. In addition, since a collaboration diagram often includes more information than a simple sequence diagram (containing information about relationships among objects), the visualization of the extracted information might be difficult. Nonetheless, collaboration diagrams provide a useful way to model the run-time behavior.

To express high-level static information, UML *component diagrams* and *packages* could be used. Similar view improvement and slicing mechanisms as discussed in this dissertation could be implemented for these views. In addition, more support for building abstractions from the low-level views could be given: the usage of these high-level models would provide natural mechanisms to produce and view the high-level information.

10.1.3 Support for iterative dynamic modeling

When synthesizing a state diagram for a participant from information given in scenario diagrams, sent events are interpreted as actions and received events as transitions. The same interpretation applies when generating a scenario diagram based on a set of interacting state diagrams. The latter process is called *tracing*.

In SCED, existing state diagrams can be used to support the construction of scenario diagrams. The user can animate the interaction of objects by using a set of collaborating state diagrams. The state diagrams simulate system behavior as objects sending events to each other and changing states according to received events. Correspondingly to the state diagram synthesis algorithm, actions are interpreted as sent events and labels of transitions as received events. The process of tracing a scenario halts when external stimuli are expected. The designer can continue the tracing process by providing that piece of information. By "guiding" the tracer through desired paths in these state diagrams, the designer produces an example sequence of interactions among corresponding objects. The result of the execution is shown as a scenario diagram. Furthermore, the designer may freely edit the traced scenario diagram at any time during the tracing process (when the tracing is halted). The scenario tracing property is hence opposite to state diagram synthesis: while the synthesis algorithm generates a state diagram from a set of scenario diagrams, the scenario tracing constructs a scenario diagram from a set of state diagrams.

SCED supports an iterative approach to construct dynamic models for object-oriented software systems semi-automatically by combining the technique for synthesizing a state diagram from information given by scenario diagrams with the tracing technique. The dynamic modeling process is smoothly changed from a "water fall" type of modeling (first scenario diagrams, then state diagrams) to a more spiral and incremental way of modeling; successive iterations increase the number of different scenario diagrams as well as refines the state diagrams. This approach is called *design-by-animation* in SCED [56, 111]. The method is especially suited for modeling the behavior of a new component using the known behavior of other, predefined, and presumably correctly implemented components. For example, such predefined components could be classes belonging to a graphical user interface framework.

The design-by-animation approach needs a set of state diagrams to start with. Such state diagrams could be produced by the dynamic reverse engineering technique described in this dissertation (cf. Chapter 8). The state diagrams of a predefined library or object-oriented framework components can be constructed by reverse engineering the run-time behavior of another system that uses the same classes. By first constructing the static dependency graph and analyzing it with Rigi, the classes of interest can be found (cf. Section 8.7). JDebugger can then be given instructions to generate event trace information especially for instances of those classes. By using SCED for viewing the event trace as scenarios, state diagrams for the objects of interest can be synthesized.

When reverse engineering the run-time behavior of a target system, the constructed state diagrams show the behavior corresponding to the usage of the system. To get complete state diagrams, the user thus needs to make sure that the usage of the objects of interest covers all possible cases. The design-by-animation approach does not require that the predefined state diagrams are complete, but they need to contain the information that will be used by the new component. If SCED is used to synthesize the predefined state diagrams, the user can be sure that the state diagrams constructed are usable for the design-by-animation approach.

10.2 Summary of contributions

The main contributions of this dissertation are as follows:

- methods for using the dependencies between static and dynamic models for goal driven reverse engineering tasks, including
  - merging dynamic information to a static Rigi view;
  - using static information to guide the generation of dynamic information;
  - slicing a Rigi view using SCED scenarios; and
  - raising the level of abstraction of SCED scenarios using a high-level Rigi graph;

- algorithms for optimizing synthesized state diagrams using UML notation;
- application of the synthesis algorithm presented by Koskimies and Mäkinen [54] to SCED;
- string matching algorithms for raising the level of abstraction of SCED scenario diagrams;
- the prototype reverse engineering environment Shimba, which integrates two existing tools:
  - Rigi for reverse engineering the static structure of Java software; and
  - SCED and its state diagram synthesis facility for reverse engineering the dynamic behavior of Java software;
- methods and tools for gathering information, including
  - extraction of static information from Java byte code; and
  - extraction of run-time information by running the target system under a customized jdk debugger;
- a case study to evaluate the facilities of Shimba.

Both static and dynamic information contains software artifacts and their relations. The shared information enables information exchange between the models. In Shimba, such a connection is used in various ways for goal-driven reverse engineering tasks. First, dynamic information can be attached to the static dependency graph (cf. Section 8.6). Weight values are given for some arcs in the dependency graph (e.g., method calls) indicating their actual usage at run-time. Furthermore, new nodes and arcs are usually added to the graph (e.g., describing exceptions thrown at run-time). Second, static information can be used to guide the generation of dynamic information (cf. Section 8.7). The Rigi view can be used to define the focus points during the debugging (i.e., the parts of the software for which event trace information is generated). Third, a Rigi graph can be sliced using a set of SCED scenario diagrams (cf. Section 8.8). This approach can be used for studying the structure of parts of the target software that are involved in a specific kind of usage. Fourth, the level of abstraction of SCED scenarios can be raised using a high-level Rigi graph (cf. Section 8.9). In Rigi, graphs can be nested to view the software on different levels of abstraction. The static abstractions built can then be used to modify SCED scenarios to view the interaction among high-level static components.

Several algorithms for optimizing a synthesized state diagram using the UML notation are presented in Chapter 6. The proposed method makes the state diagram more readable and compact, yet preserving its information content. The algorithms detect similar responses to certain events and use that information to restructure the state diagram. The state diagram is modified by adding UML statechart notation elements into it.

An algorithm for synthesizing state machines automatically from trace diagrams is presented by Koskimies and Mäkinen [54]. This dissertation presents an application of that algorithm to state diagram synthesis in SCED (cf. Chapter 5). The scenario notation of SCED is richer than the basic MSC notation used in [54]. In addition, some rules concerning join of states during the synthesis have been added.

String matching algorithms can be applied to SCED scenarios for searching behavioral patterns. The patterns found provide means to raise the level of abstraction of the scenarios and to decrease their size (cf. Section 8.5). The SCED scenario diagram notation includes means to visualize those patterns.

A prototype environment Shimba was built for reverse engineering Java systems (cf. Chapter 8). The static information is viewed as a directed graph using Rigi. A new Java domain model (cf. Appendices A-C) that enables Rigi to visualize the target system was produced. In addition, several scripts were written to help the user to analyze the dependency graph using Rigi. The dynamic information in Shimba is visualized as scenario diagrams using SCED. An overall run-time behavior of a single object or a method can be visualized as a state diagram using the automatic state diagram synthesis facility of SCED.

Both static and dynamic information is extracted from Java byte code (cf. Section 8.2). Types of software artifacts and their dependencies read from Java class files are included in the Java domain model (cf. Appendices A-C). The dynamic information is generated by running the target application or applet under a customized jdk debugger. The event trace information is saved in a format readable for SCED. In addition to member function calls, dynamic control flow information, which describes branching within member functions, can be generated (cf. Section 8.4).

To validate the usefulness of the reverse engineering approach described in this dissertation, a moderate size target Java software system was examined (cf. Chapter 9). The selected target system FUJABA (version 0.6.3-0) contains almost 700 classes. In the case study, we used both static and dynamic reverse engineering techniques of Shimba, as well as all the techniques of exchanging information between the views.

10.3 Directions for future work

The UML notation contains several diagram types that can be used to model a system from different perspectives and on 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 on 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.

The use of transformation and synthesis mechanisms is not limited between two diagram types. A more refined model can usually be constructed if the information is gathered from different sources. For example, a UML collaboration diagram could be (automatically) constructed from sequence and object diagrams. Furthermore, 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 the future work.

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. A reverse engineering environment that uses the different UML diagram types would be desirable. The current implementation of Shimba does not support the full UML notation. Possibilities to integrate the reverse engineering techniques of Shimba in an object-oriented modeling tool TED [121] will be a part of the future work. TED has been implemented at Nokia Research Center and is currently used at Nokia. TED supports UML sequence diagram, statechart diagram, class diagram, object diagram, collaboration diagram, use case diagrams, and implementation diagram notations.

10.4 Concluding remarks

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 the software 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 the software. The reverse engineering technique proposed in this dissertation was developed to help the engineer to reach this goal.

Bibliography

[1] Advanced Software Technologies Inc., *ASTI - Welcome to GDPro!*, http://www.advancedsw.com/welcome.html, 1999.

[2] Aho A., Sethi R., and Ullman J., *Compilers: Principles, Tools and Techniques*, Addison-Wesley 1986.

[3] Angluin D., Computational learning theory: survey and selected bibliography, In *Proc. 24th Ann. ACM Symp. on the Theory of Computing*, 1992, pp. 351–369.

[4] Angluin D. and Smith C.H., Inductive inference: theory and methods, *ACM Comput. Surv.*, **15**, 3, 1983, pp. 237–269.

[5] Bansiya J., Automating Design-Pattern Identification, *Dr. Dobb's Journal*, http://www.ddj.com/, 2000.

[6] Biermann A.W., Baum, R.I., and Petry, F.E., Speeding up the Synthesis of Programs from Traces, *IEEE Trans. on Computers*, **24**, 2, 1975, pp. 122–136.

[7] Biermann A.W. and Krishnaswamy, R., Constructing programs from example computations, *IEEE Trans. Softw. Eng.*, **2**, 3, 1976, pp. 141–153.

[8] Biggerstaff T.J., Design recovery for maintenance and reuse, *IEEE Software*, **22**, 7, July, 1989, pp. 36–49.

[9] Booch G., *Object-Oriented Analysis and Design with Applications*, 1st ed., Benjamin Cummings, 1991.

[10] Booch G., *Object-Oriented Analysis and Design with Applications*, 2nd ed., Benjamin Cummings, 1994.

[11] Booch G., Rumbaugh J., Jacobson I., *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.

[12] Boyer R. S. and Moore J. S., A Fast String Searching Algorithm, *Comm. of the ACM*, **20**, 10, 1997, pp. 762–772.

[13] CCITT, Document COM X-R 33-E, New Recommendation Z.120, Message Sequence Charts, July 1992.

[14] Chase M., Christey S., Harris D., and Yeh A., Managing Recovered Function and Structure of Legacy Software Components, In *Proc. of the 5th Working Conference on Reverse Engineering (WCRE98)*, IEEE Computer Society Press, 1998, pp. 79–88.

[15] Chen Y.-F., Fowler G., Koutsofios E., and Wallach R., Ciao: A Graphical Navigator for Software and Document Repositories, In *Proc. of the International Conference on Software Maintenance (ICSM95)*, Nice, France, October 1995, pp. 66–75.

[16] Chidamber S.R. and Kemerer C.F., Towards a Metrics Suite for Object-Oriented Design, In *Proc. of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'91)*, ACM Press, 1991, pp. 197–211.

[17] Chidamber S.R. and Kemerer C.F., A Metrics Suite for Object- Oriented Design, *IEEE Trans. Softw. Eng.*, **20**, 6, 1994, pp. 476–493.

[18] Chikofsky E. and Cross J., Reverse Engineering and Design Recovery: A Taxonomy, *IEEE Software*, **7**, 1, January 1990, pp. 13–17.

[19] Coad P. and Yourdon E., *Object-Oriented Analysis*, 2nd ed., Yourdon Press, 1991.

[20] Coleman D., Arnold P., Bodoff S., Dollin C., Gilchrist H., Hayes F., and Jeremaes P., *Object-Oriented Development: The Fusion Method*, Prentice Hall, 1994.

[21] Coleman D., Hayes F., and Bear S., Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design, *IEEE Trans. Softw. Eng.*, **18**, 1, January 1992, pp. 9–18.

[22] Computer Associates International, *Computer Associates Homepage*, http://www.cai.com/, 1999.

[23] CS Verilog, *CS Verilog Homepage*, http://www.verilogusa.com/, 1999.

[24] Damm W. and Harel D., LSCs: Breathing Life into Message Sequence Charts, In *IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99)*, Kluwer Academic Publishers, 1999, pp. 293–312.

[25] DeMarco T., *Controlling Software Projects*, Yourdon Press, 1982.

[26] De Pauw W., Helm R., Kimelman D., and Vlissides J., Visualizing the Behavior of Object-Oriented Systems, In *Proc. of the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93)*, ACM Press, 1993, pp. 326–337.

[27] De Pauw W., Lorenz D., Vlissides J., and Wegman M., Execution Patterns in Object-Oriented Visualization, In *Proc. of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, Santa Fe, New Mexico, April, 1998, pp. 219-234.

[28] Demeyer S., Ducasse S., and Lanza M., A Hybrid Reverse Engineering Approach Combining Metrics and Program Visualization, In *Proc. of the 6th Working Conference on Reverse Engineering*, IEEE Computer Society Press, 1999, pp.175–186.

[29] Domain Objects Inc., http://www.domain-objects.com/, 1999.

[30] Dósa F. and Koskimies K., Tool-Supported Compression of UML Class Diagrams, In *Proc. of the 2nd International Conference on the Unified Modeling Language (UML'99)*, Springer-Verlag 1999, pp. 172-187.

[31] Ellis M. and Stroustrup B., *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[32] Even S., *Graph Algorithms*, Pitman, 1979.

[33] Expert software systems, *E2S, A Software Engineering and Case Tool Company*, http://www.e2s.be/, 1999.

[34] Finnigan P., Holt R., Kalas I., Kerr S., Kontogiannis K., Müller H., Mylopoulos J., Perelgut S., Stanley M., and Wong K., The software bookshelf, *IBM Systems Journal*, **36**, 4, 1997, pp 564–593.

[35] Fowler M., *Refactoring*, Addison-Wesley, 1999.

[36] Gamma E., Helm R., Johnson R., and Vlissides J., *Design Patterns: Elements of Object-Oriented Software Architecture*, Addison-Wesley, 1995.

[37] Garey M.R. and Johnson D.S., *Computers and Intractability: A Guide to the Theory of NP-completeness*, Freeman, 1979.

[38] Glinz M., An Integrated Formal Model of Scenarios Based on Statecharts, *Lecture Notes in Computer Science 989*, Springer-Verlag, 1995, pp. 254–271.

[39] Graham I.M., *Migrating to Object Technology*, Addison-Wesley, 1995.

[40] Harel D., Statecharts: A Visual Formalism for Complex Systems, *Science of Computer Programming*, **8**, 1987, pp. 231–274.

[41] Harel D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., Shtull-Tauring, A., and Trakhtenbrot, M., STATEMATE: A Working Environment for the Development of Complex Reactive Systems, *IEEE Trans. Softw. Eng.*, **16**, 4, 1990, pp. 403–414.

[42] Harel D. and Politi M., *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*, McGraw-Hill, 1998.

[43] Henderson-Sellers B., *Object-Oriented Metrics, Measures of Complexity*, Prentice Hall, 1995.

[44] Hsia P., Samuel J., Gao J., Kung D., Toyoshima Y., and Chen C., Formal Approach to Scenario Analysis, *IEEE Software*, **11**, 2, March 1994, pp. 33–41.

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

[46] Imagix Corporation, *Imagix - Reverse engineering tools*, http://www.imagix.com/index.html, 1999.

[47] Innovative Software GmbH, *Innovative Software Homepage*, http://www.innovative-software.co.uk/oew/index.html, 1999.

[48] IntegriSoft Inc., *IntegriSoft Homepage*, http://www.integrisoft.com/, 1999.

[49] Z.120 ITU-T Recommendation Z.120: Message Sequence Chart (MSC), ITU-T, Geneva, 1996.

[50] Jacobson I., *Object-Oriented Software Engineering — A Use Case Driven Approach*. Addison-Wesley, 1992.

[51] Jerding D. and Rugaber S., Using Visualization for Architectural Localization and Extraction, In *Proc. of the 4th Working Conference on Reverse Engineering (WCRE97)*, IEEE Computer Society Press, 1997, pp. 56–65.

[52] Kazman R. and Carriere J., Playing Detective: Reconstructing Software Architecture from Available Evidence, *Automated Software Engineering*, **6**, 2, 1999, pp. 107–138.

[53] Khriss I., Elkoutbi M., and Keller R., Automating the Synthesis of UML Statechart Diagrams from Multiple Collaboration Diagrams, In Bezivin J. and Alain P. (eds.), *The Unified Modeling Language. UML'98: Beyond the Notation*, Springer-Verlag, LNSC 1618, 1999, pp. 132-147.

[54] Koskimies K. and Mäkinen E., Automatic Synthesis of State Machines from Trace Diagrams, *Softw. Pract. & Exper.*, **24**, 7, 1994, pp. 643–658.

[55] Koskimies K., Männistö T., Systä T., and Tuomi J., On The Role of Scenarios in Object-Oriented Software Design, In *Proc. of the Nordic Workshop on Programming Environment Research (NWPER'96)*, Aalborg, Institute of Electronic Systems, Aalborg University, Denmark, 1996, pp. 53–70.

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

[57] Koskimies K., Männistö T., Systä T., and Tuomi J., SCED: A Tool for Dynamic Modelling of Object Systems, University of Tampere, Dept. of Computer Science, Report A-1996-4, 1996.

[58] Koskimies K., Männistö T., Systä T., and Tuomi J., SCED — An Environment for Dynamic Modeling in Object-Oriented Software Construction, In *the Proc. Nordic Workshop on Programming Environment Research (NWPER'94)*, Lund, Department of Computer Science, Lund Institute of Technology, Lund University, 1994, pp. 217–230.

[59] Koskimies K. and Mössenböck H., Scene: Using Scenario Diagrams and Active Text for Illustrating Object-Oriented Programs, In *Proc. International Conference on Software Engineering (ICSE '96)*, ACM Press, 1996, pp. 366–375.

[60] Lange D. B. and Nakamura Y., Interactive Visualization of Design Patterns Can Help in Framework Understanding, In *Proc. of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, ACM Press, 1995, pp. 342–357.

[61] Lange D. B. and Nakamura Y., Object-Oriented Program Tracing and Visualization, *IEEE Computer*, **30**, 5, 1997, pp. 63-70.

[62] Leue S., Mehrmann L., and Rezai M., Synthesizing ROOM Models from Message Sequence Chart Specifications, University of Waterloo, Tech. Report 98-06, 1998.

[63] Leue S., Mehrmann L., and Rezai M., Synthesizing Software Architecture Descriptions from Message Sequence Chart Specification, In *Proc. of the 13th IEEE International Conference on Automated Software Engineering (ASE98)*, IEEE Computer Society Press, 1998, pp. 192–195.

[64] Li W. and Henry S., Maintenance Metrics for the Object Oriented Paradigm, In *Proc. of the 1st Intl. Software Metrics Symposium*, IEEE Computer Society Press, 1993, pp. 52-60.

[65] Li W. and Henry S., Object-Oriented Metrics that Predict Maintainability, *Journal of Systems and Software*, **23**, 1993, pp. 111–122.

[66] Lorenz M. and Kidd J., *Object-Oriented Software Metrics, A Practical Guide*, Prentice Hall, 1994.

[67] Männistö T., Systä T., and Tuomi J., SCED Report and User Manual, University of Tampere, Dept. of Computer Science, Report A-1994-5, 1994.

[68] Männistö T., Systä T., and Tuomi J., Design of State Diagram Facilities in SCED, University of Tampere, Dept. of Computer Science, Report A-1994-11, 1994.

[69] Männistö T., Systä T., and Tuomi J., Synthesizing OMT State Diagrams, In *Proc. of the 4th Symposium on Programming Languages and Software Tools*, Dept. of Computer Science, Eötvös Lórand University of Budapest, 1995, pp. 21–32.

[70] McCabe T.J.,A Complexity Measure, *IEEE Trans. Softw. Eng.*, **2**, 4, 1976, pp. 308–320.

[71] McCabe & Associates Inc., *McCabe & Associates — Making IT Right!*, http://www.mccabe.com/, 1999.

[72] Murphy G., Notkin D., Griswold W., and Lan E., An Empirical Study of Static Call Graph Extractors, *ACM Trans. Softw. Eng. Methodol.*, **7**, 2, 1998, pp. 158–191.

[73] Müller H., Orgun M., Tilley S., and Uhl J., A Reverse-engineering Approach to Subsystem Structure Identification, *Software Maintenance: Research and Practice*, **5**, 1993, pp. 181–204.

[74] Müller H., Wong K., and Tilley S., Understanding Software Systems Using Reverse Engineering Technology, In *Proc. of the 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences Proceedings (ACFAS)*, 1994.

[75] Näher S., *LEDA User Manual, Version 3.0*. Max-Planck-Institut für Informatik, 1992.

[76] Ousterhout J.K., *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.

[77] Paakki J., Salminen A., and Koskinen, J., Automated hypertext support for software maintenance, *The Computer Journal*, **39**, 7, 1996, pp. 577-597.

[78] Portner N., Flexible Command Interpreter: A Pattern for an Extensible and Language-Independent Interpreter System, In Coplien J., Schmidt D. (eds.), *Pattern Languages of Program Design*, Addison-Wesley, 1995, pp. 43–50.

[79] Rational Software Corporation, *Version 0.8 of the Unified Method*, http://www.rational.com/ot/uml/0.8/index.html, October 1995.

[80] Rational Software Corporation, *Version 0.91 of the Unified Modeling Language*, http://www.rational.com/ot/uml/0.91/uml91.pdf, September 1996.

[81] Rational Software Corporation, *Version 1.0 of the Unified Modeling Language*, http://www.rational.com/ot/uml/1.0/index.html, January 1997.

[82] Rational Software Corporation, *Rational Rose 98: Using Rational Rose*, 1998.

[83] Rational Software Corporation, *Rational Rose 98: Roundtrip Engineering with C++*, 1998.

[84] Rational Software Corporation, *Rational Rose 98: Roundtrip Engineering with Java*, 1998.

[85] Rational Software Corporation, *The Unified Modeling Language Notation Guide v.1.3*, http://www.rational.com, January 1999.

[86] Reasoning Inc., *Reasoning - Software Enhancement e-Services*, http://www.reasoning.com/, 1999.

[87] Richner T. and Ducasse S., Recovering High-Level Views of Object-Oriented Applications form Static and Dynamic Information, In *Proc. of the International Conference on Software Maintenance (ICSM99)*, IEEE Computer Society Press, 1999, pp. 13–22.

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

[89] Rugaber S., A Tool Suite for Evolving Legacy Software, In *Proc. of the International Conference on Software Maintenance (ICSM99)*, IEEE Computer Society Press, 1999, pp. 33-39.

[90] Rumbaugh J., Series on 2nd Generation OMT, *JOOP*, **7** and **8**, 1994–1995.

[91] Rumbaugh J., OMT: The Dynamic Model, *Journal of Object-Oriented Programming*, **7**, 9, 1995, pp. 6–12.

[92] Rumbaugh J., OMT: The Functional Model, *Journal of Object-Oriented Programming*, **8**, 1, March/April 1995, pp. 10–14.

[93] Rumbaugh J., OMT: The Development Process, *Journal of Object-Oriented Programming*, A SIGS Publication, **8**, 2, May 1995, pp. 8–16.

[94] Rumbaugh J., Blaha M., Premerlani W., Eddy F., and Lorensen W., *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

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

[96] Schönberger S., Keller R., and Khriss I., Algorithmic Support for Transformations in Object-Oriented Software Development, Technical Report GELO-83, University of Montreal, 1998.

[97] Schönberger S., Keller R., and Khriss I., Algorithmic Support for Model Transformation in Object-Oriented Software Development, In *Theory and Practice of Object Systems (TAPOS)*, John Wiley & Sons, 1999, to appear.

[98] Shlaer S. and Mellor S.J., *Object-Oriented Systems Analysis: Modeling the World in Data*, Yourdon Press, 1988.

[99] Sefika M., Sane A., and Campbell R.H., Architecture-Oriented Visualization, In *Proc. of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96)*, ACM Press, 1996, pp. 389–405.

[100] Selic B., Gullekson G., and Ward P., *Real-Time Object-Oriented Modeling*, John Wiley & Sons, 1994.

[101] Smart J., *wxWindows Home*, http://web.ukonline.co.uk/julian.smart/wxwin/, 1999.

[102] Somé S. and Dssouli R., An Enhancement of Timed Automata Generation from Timed Scenarios Using Grouped States, Université de Montréal, DIRO Technical Report #1029, April 1996.

[103] Somé S., Dssouli R., and Vaucher J., From Scenarios to Automata: Building Specifications from Users Requirements, APSEC'95, IEEE Computer Society Press, 1995, pp. 48–57.

[104] Somé S., Dssouli R., and Vaucher J., Towards an Automation of Requirements Engineering using Scenarios, *Journal of Computing and Information*, **2**, 1, 1996, pp. 1110–1132.

[105] Sterling Software Inc., *Welcome to Sterling Software*, http://cool.sterling.com/, October, 1999.

[106] Storey M.-A.D., A Cognitive Framework For Describing and Evaluating Software Exploration Tools, PhD Dissertation, Technical Report, School of Computing Science, Simon Fraser University, December 1998.

[107] Storey M.-A.D., Wong K., and Müller H., Rigi: A Visualization Environment for Reverse Engineering, In *Proc. of the International Conference on Software Engineering (ICSE'97)*, Boston, U.S.A., 1997, pp. 606–607.

[108] Stroustrup B., *The Annotated C++ Reference Manual*, Addison-Wesley, 2nd ed., 1991.

[109] Stroustrup B., *The Annotated C++ Reference Manual*, Addison-Wesley, 3rd ed., 1997.

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

[111] Systä T., Incremental Construction of OMT Dynamic Model, *Journal of Object-Oriented Programming*, to appear.

[112] Systä T., On the Relationships between Static and Dynamic Models in Reverse Engineering Java Software, In *Proc. of the 6th Working Conference on Reverse Engineering (WCRE99)*, IEEE Computer Society Press, 1999, pp.304–313.

[113] Systä T. and Yu P., Using Object-Oriented Metrics and Rigi to Evaluate Java Software, University of Tampere, Dept. of Computer Science, Report A-1999-9, July, 1999.

[114] Systä T., Yu P., and Müller H., Analyzing Java Software by Combining Metrics and Program Visualization, In *Proc. of the 4th European Conference on Software Maintenance and Reengineering (CSMR2000)*, to appear.

[115] TakeFive Software Inc., *TakeFive Software Homepage*, http://www.takefive.com/, 1999.

[116] , Tilley S. and Müller H., Using Virtual Subsystems in Project Management, In *Proc. of the IEEE Sixth International Conference on Computer-Aided Software Engineering(CASE)*, 1993, pp. 144–153.

[117] Tilley S., Wong K., Storey M.-A., and Müller H., Programmable Reverse Engineering, *International Journal of Software Engineering and Knowledge Engineering*, **4**, 4, 1994, pp. 501–520.

[118] Venners B., *Inside the Java Virtual Machine*, McGraw-Hill, 1998.

[119] Viasoft Inc., *Viasoft Home Page*, http://www.viasoft.com/, 1999.

[120] Walker R., Murphy G., Freeman-Benson B., Wright D., Swanson D., and Isaak J., Visualizing Dynamic Software System Information through High-level Models, In *Proc. of the 1998 ACM Conference on Object-Oriented Programming, Systems, Languages, and Application (OOPSLA'98)*, ACM Press, 1998, pp. 271-283.

[121] Wikman J., *Evolution of a Distributed Repository-Based Architecture*, http://www.ide.hk-r.se/∼bosch/NOSA98/JohanWikman.pdf, 1998.

[122] Wirfs-Brock R., Wilkerson B., and Wiener L., *Designing Object-Oriented Software*, Prentice Hall, 1990.

[123] Wong K., *Rigi User's Manual Version 5.4.1*, http://www.rigi.csc.uvic.ca/rigi/manual/user.html, September, 1997.

Appendix A

Rigi domain model for Java: Riginode file

Collapse

♯ JExtractor does not currently produce System nodes

System

♯ JExtractor does not currently produce Release nodes

Release

♯ JExtractor does not currently produce Revision nodes

Revision

Composite

Class

Method

```
Constructor


♯ All Variable nodes refer to class variables
♯ Information about local variables is not extracted
Variable


Interface


♯ Staticblock is used for initializing static class variables
Staticblock


♯ Exceptions are in fact classes, though dynamically they
♯ have a specific role.
♯ Hence it seems to be desirable to have a different node
♯ representing exceptions
Exception


Unknown
```

Appendix B

Rigi domain model for Java: Rigiarc file

```
♯ calls are method or constructor invocations, i.e.
♯ call arc can be between two Method nodes, between a
♯ Method node and a Constructor node and between two
♯ Constructor nodes.
♯ Dynamically (debugged infromation) there is always also
♯ call arcs from/to a static block if a class defines one.
call

♯ inherit arcs (extend clause in Java) can be between two
♯ Class nodes or between two Interface nodes.
inherit

♯ implement arc is always from a Class node to an Interface node
implement Class Interface

♯ following cases are possible containment relationships defined
♯ with a contain arc:
♯ contains Class Method
♯ contains Class Variable
♯ contains Class Statickblock
♯ contains Class Class
♯ contains Class Constructor
♯ contains Interface Method
♯ contains Interface Variable
contains

♯ throw arcs are generated when an exception is thrown.
♯ Currently these arcs are generated only during run-time.
♯ representing dynamic information only.
♯ If throw arcs were generated for all methods and classes
♯ that can throw an exception, the number of them would be huge.
♯ So far, there hasn’t been any need or reason to do this, but
♯ it might be worth considering in the future.
throw

♯ access arcs represent class variable (Variable nodes) usage.
♯ following cases are possible:
♯ access Method Variable
♯ access Constructor Variable
♯ access Staticblock Variable
access

♯ assign arcs represent class variable (Variable nodes) assignments,
♯ i.e. the value of the variable is changed.
♯ following cases are possible:
♯ assign Method Variable
♯ assign Constructor Variable
♯ assign Staticblock Variable
```

```
assign


♯ composite arcs are created by when running some rcl scripts.
♯ They represent high level arcs and are used if either end
♯ of the arc is a high level Collapse node
composite


♯ level arcs are generated by Rigi. They represent
♯ subsystem hierarchies and are used in structured rsf
level
```

Appendix C

Rigi domain model for Java: Rigiattr file

```
♯
♯ node attributes:
♯

♯ package attributes are generated only for Classes and
♯ Interfaces. There is no need to generate package values
♯ for Methods, Variables, etc. since they are always
♯ encaptulated inside Classes or Interfaces.
♯ The value is a string representing the package name.
♯ Though, there is not really any use for this attribute
♯ because the Class and Interface nodes have long names
♯ including the package name: e.g., java.io.InputStream
attr Node package

♯ visibility, static, abstract, native, final, and volatile
♯ attributes are generated for Classes, Interfaces, Methods,
♯ Constructors, Staticblocks, and Variables
♯ The value is either public, protected, or private
attr Node visibility

♯ The value is 1 (is static) or 0 (is not static)
attr Node static

♯ Thevalue is 1 (is abstract) or 0 (is not abstract)
attr Node abstract

♯ The value is 1 (is native) or 0 (is not native)
attr Node native

♯ The value is 1 (is final) or 0 (is not final)
attr Node final

♯ The value is 1 (is volatile) or 0 (is not volatile)
attr Node volatile

♯ The value is 1 (is synchronized) or 0 (is not synchronized)
attr Node synchronized

♯ The value of lineno attribute is a string representing
♯ the name of the file the Class or Interface is
♯ located in.
♯ filename attribute is generated for Class and Interface
♯ nodes only.
attr Node filename
```

```
♯ The value is an integer representing the line number in
♯ the source file the Method, Constructor, Staticblock, or
♯ Variable is defined at.
♯ For Methods, Constructors, and Staticblocks it is the
♯ the first line.
♯ This is still under implementation
attr Node lineno

♯ url attribute is currently not used.
attr Node url

♯ annotation attribute is currently not used
attr Node annotation

♯ The value is a string representing a path+file name
♯ where the javadoc documentation exist.
♯ The htmlized documentation can be viewed using, e.g., any
♯ web browser (java_show_documentation script)
♯ Thes values can be generated by the Ideogram environment
attr Node documentation

♯ The value of return attribute is a string representing the
♯ return type of a node.
♯ return attribute values are generated for Classes, Interfaces,
♯ Methods, Constructors, and Variables
attr Node return

♯ Next 9 attributes represent OO metrics values that can
♯ be generated for Classes, Interfaces, Methods, and
♯ Constructors using the JMetricsProgram and/or Ideogram environment
♯ The values are integers.
♯ LOC: Lines of Code (under implementation)
attr Node LOC

♯ DIT: Depth of Inheritance Tree
attr Node DIT

♯ NOC: Number of Children
attr Node NOC

♯ CC: McCabe's Cyclomatic Complexity
attr Node CC

♯ CBO: Coupling Between Objects
attr Node CBO

♯ LCOM: Lack of Cohesion in Methods
attr Node LCOM

♯ WAC: Weighted Attributes per Class (used LOC, under implementation)
attr Node WAC

♯ WMC: Weighted Methods per Class
attr Node WMC

♯ RFC: Response For a Class
attr Node RFC
```

```
♯
♯ arc attributes:
♯

♯ filename attribute values are not currently generated
attr Arc filename

♯ lineno attributes are not currently generated
attr Arc lineno

♯ url attributes are not currently generated
attr Arc url

♯ annotation attributes are not currently generated
attr Arc annotation

♯ The actual weight attribute values are generated at run-time.
♯ They represent the number of times the arc is actually used.
♯ weight values are generated for call, access, assign, and throw
♯ arc.
♯ The default values (static value) is 0.
♯ When dynamic information is also included, the value of
♯ weight attribute is stored in comments:
♯ e.g., '♯♯ call myPackage1.myCl1.foo myPackage2.myCl2.foo() weight 3'
♯ The reason is that basic unstructured rsf files consist of triples
♯ but the arc attributes need four tokens (sender received attr attrValue)
♯ Files with such dynamic information can be read using java_load script.
attr Arc weight
```

Appendix D

Calculating software metrics in Shimba

Depth of Inheritance Tree (DIT)

For a class or an interface, DIT is the number of its ancestor classes or interfaces. Foundation classes (jdk) are ignored.

Number of Children (NOC)

For a class, NOC is the number of classes that extend this class. For an interface, NOC is the sum of the following two values: the number of interfaces that extend the interface and the number of classes that implement it. The foundation classes (jdk) are ignored.

Response For a Class (RFC)

For a class C, let M_i be the set of all member functions in C. Let M_o be the set of all member functions, belonging to other classes, that are called by the members of M_i. Then RFC is the size of the set $M_i \cup M_o$.

Coupling Between Objects (CBO)

The following dependencies between two classes, which are not in a superclass-subclass relationship, constitute coupling that is counted when calculating CBO: method calls, constructor calls, instance variable assignments, or other kind of instance variable accesses.

Lack of Cohesion in Methods (LCOM)

In Shimba, we calculate LCOM using the formula that has been presented by Henderson-Sellers [43]. For a class C, let M be a set of its m methods $M_1, M_2, \ldots M_m$, and let A be a set of its a data members $A_1, A_2, \ldots A_a$ accessed by M. Let $\mu(A_k)$ be the number of methods that access data attribute A_k where $1 \leq k \leq a$. Then $LCOM(C(M,A))$ is defined as follows:

$$LCOM(C(M,A)) = \frac{\left(\frac{1}{a}\sum_{j=1}^{a} \mu(A_j)\right) - m}{1 - m} \tag{D.1}$$

Cyclomatic Complexity

In Shimba, we use the following formula, adopted from Henderson-Sellers [43], to compute CC:

$$CC(G) = e - n + 2p, \tag{D.2}$$

where G is a complexity graph, n and e are the number of nodes and edges in G, respectively, and p is the number of disconnected components in G. The complexity graph G for a single method is a control flow graph.

Weighted Methods per Class (WMC)

WMC is defined as the sum of the complexities of all the methods of a class except the inherited methods but including overloaded methods. The Henderson-Seller Cyclomatic Complexity CC [43] is used to compute the complexity of a method:

$$WMC = \sum_{i=1}^{n} CC_i \tag{D.3}$$