

# Extracting state diagrams from legacy systems

Tarja Systä and Kai Koskimies

Department of Computer Science, University of Tampere, P.O. Box 607, FIN-33101

Tampere, Finland

{cstasy, koskimie}@cs.uta.fi

## 1. Introduction

A basic problem of reverse engineering is to understand legacy systems and derive abstract characterizations of poorly documented software. In the case of object-oriented software, the static structure (e.g. inheritance and association relationships) can usually be understood easily, and it can be extracted from existing software using automated tools. This is due to the fact that the static aspects of object-oriented programs are well known and more or less explicitly indicated in the source. Understanding and characterizing the dynamic behavior of such systems, in contrast, is usually much more difficult because of the gap between the static source text and the run-time behavior of the resulting executable program. However, the dynamic behavior of a program is equally important as its static specification for understanding the software. Dynamic characterization is particularly important for those parts of a system which are mainly understood by their dynamic behavior, like various kinds of controllers, drivers etc.

For instance, assume that the control unit of an elevator needs to be re-engineered, and that this software lacks all documentation. We can identify certain classes in the source that presumably represent various kinds of controller objects, but on the basis of the source we can only observe that they react on certain kinds of events in a particular way, depending on their current state, and that they in turn send certain events to other objects. However, it is extremely difficult to infer the general behavior of such objects in the form of, say, state diagram on the basis of static information only. The situation is somewhat better if the source is systematically written to reflect the structure of a state machine, but this is often not the case.

In this paper we demonstrate that it is possible to solve this problem by combining two existing techniques: the production of scenario diagrams from running systems, and the synthesis of state diagrams from a set of scenario diagrams. Scenario diagrams (sometimes called interaction diagrams or event trace diagrams) are a popular graphical notation for describing the interactions of a set of objects and actors during a particular usage of a system. Scenario diagrams are traditionally used as analysis and design documents (e.g. [Rum91], [Jaa97]), but recently they have been used also for animating or visualizing the dynamic behavior of a running system ([KoM96], [LaN95], [SSC96]). In this paper, the visualization of the run-time behavior of object-oriented programs is taken one step further: not only scenarios 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 the technique recently developed for automatically synthesizing state diagrams from scenario diagrams ([KoM94], [KMST96]).

Roughly, our solution works in practice as follows. The source program is first instrumented with code that registers the events and conditions concerning interesting objects. We assume that the instrumentation can be done by an automated tool; however, for the purposes of this paper we have done it by hand. The event/condition sequence is produced at the run-time on the basis of the instrumentation code and fed to an existing system, SCED [KMST96], using a small intermediate program, called

Program Tracer. This program merely analyzes the event/condition sequence and sends it further to SCED in a proper format. SCED, in turn, constructs scenario diagrams describing the interaction of a set of objects implied by the event/condition sequence. At any point, SCED can be asked to synthesize the general behavior of an object as a (minimal) state diagram, given a set of scenario diagrams in which the object participates. Thus, we can run the instrumented target system with various input for a while, and at the same time observe the interaction of the interesting objects as scenario diagrams. Then we can select one of the objects, and ask the system to produce a state diagram for it. Depending on how covering the input was, a more or less complete state diagram is shown by the system. Figure 1 shows the different phases of the process of extracting a state machine from a source code.

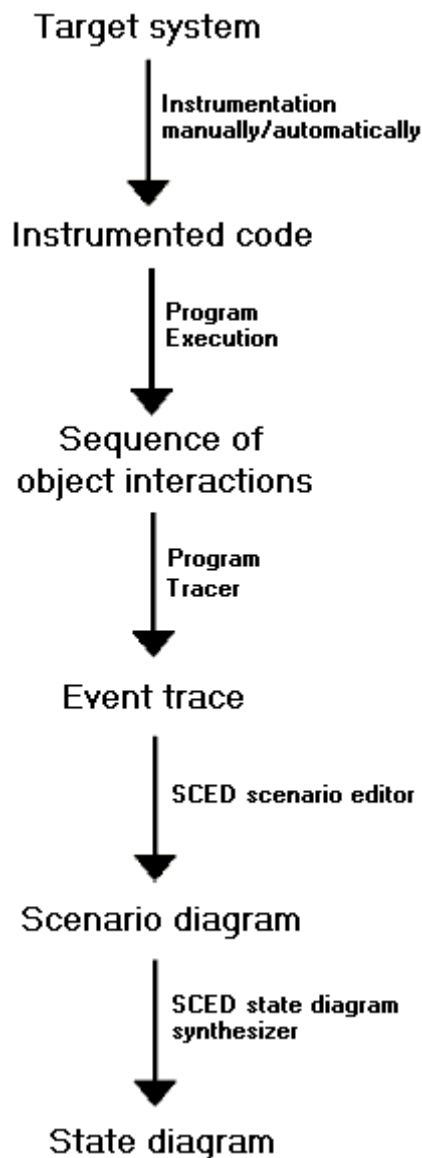


Figure 1. The procedure for visualizing the run-time behavior of a target system as a state diagram.

In section 2 we briefly explain the method we are using for synthesizing a state diagram on the basis of information given by scenarios. The instrumentation of the target system is discussed in section 3. Section 4 introduces a case study: the behavior of a desk calculator (Java applet) is visualized following the procedure described in figure 1. Finally, in section 5 some ideas for further elaboration of the presented approach are discussed.

## 2. Synthesizing state diagrams

SCED [KMST96] has been built to raise the level of automated support for the dynamic modeling part of object-oriented software development. It uses OMT [Rum91] method as a guideline and notational basis, but in principle the system is not tied with any particular methodology. Although SCED is primarily intended to support traditional OMT-style object-oriented software development, it can be used for other scenario driven modeling purposes as well. In fact, in the experimental system presented in this paper, SCED is used for opposite purposes with respect to its normal use: it is used for reverse rather than forward engineering. For more detailed description of the system the reader is referred to [KMST96].

In SCED, the rather rudimentary scenario diagram notation of OMT has been extended in various ways. For presenting other primitive actions than events, *an action box* is adopted to SCED scenario notation. An action box is also used to denote events received by the sender itself, e.g. calls for object's own methods. Another extension is *an assertion box*, which can be used for expressing conditions that are known to hold at a certain position in a scenario for a particular object. We omit here the rest of the extensions since they are not essential for the purposes of the present paper. Figure 2 shows a SCED scenario.

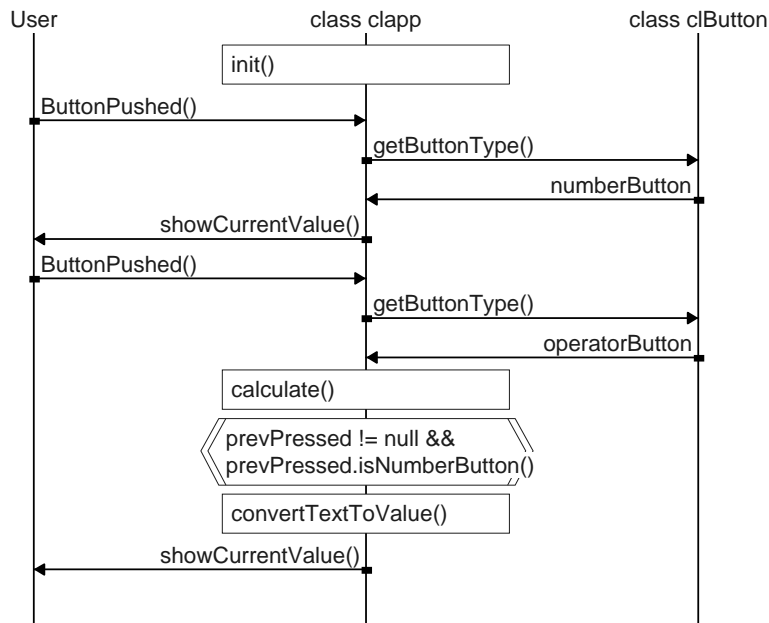


Figure 2. A SCED scenario.

A basic observation behind SCED is that constructing scenarios and fusing them into a state diagram can be supported by automated tools far more than what is done in current systems. The algorithm used for the state diagram synthesis was based on a method for synthesizing programs from their example traces originally presented by Biermann and Krishnaswamy in [BiK76]. The Biermann-Krishnaswamy (BK) algorithm was used in an actual system that was able to synthesize programs on the basis of examples of sequences of primitive actions (like assignments) and assertions given by the programmer using a partly graphical interface. Essentially, the user gives traces (i.e. sequences of such actions and assertions) of the expected program, and the algorithm produces the smallest program that is capable of executing the given example traces.

Due to the similarities between the concepts of a program (as used in BK -algorithm) and a state diagram, and on the other hand, between concepts of a trace and a scenario, the BK-algorithm can be applied to state diagram synthesis as well. A trace is extracted from a scenario by selecting a participating object (i.e. the object for which the state machine will be synthesized), and traversing the vertical line of that object from top to bottom. Each received event is regarded as an assertion (‘event  $e$  has occurred’) and each sent event is regarded as a primitive action (‘cause event  $e$ ’) in terms of the BK-algorithm. First, a lower bound  $N$  for the number of states of the resulting state machine is required. For this we use the number of different actions: since each state can have at most one action, this is clearly a lower bound. The algorithm maps actions to states and assertions to transitions, starting from the beginning of the trace. If a nondeterministic state results, the algorithm backtracks to a previous position where there was some freedom in associating an action with a state, and takes another untried choice. If at some point  $N+1$  states are needed, the algorithm backtracks again. If backtracking is no more possible, a state machine with  $N$  states cannot be achieved. Then the whole process is repeated for the allowed number of states  $N+1$  etc. The algorithm is completed when all actions have been associated with states. The BK-algorithm works incrementally, too: a scenario can be fused into an existing state machine using the algorithm. Both assertion boxes and action boxes can be incorporated in the synthesis algorithm described above in a simple way: an assertion is regarded as an event without a sender, an action is regarded as an event without a receiver.

Another system for generating state machines from example scenarios have been introduced by Somé et al in [SDV95]. They have devised an incremental algorithm, used in implemented tools, for generating timed-automata from scenarios. Hsia et al, in turn, present a formalization of scenarios as *scenario trees*, from which an abstract state machine can be constructed [FASA94].

SCED offers various tools for modifying the synthesized state diagram. For instance, the state diagram can be simplified, still preserving its information, by generating advanced OMT state diagram notation for it, i.e. by attaching actions with transitions, by generating entry and exit actions for states etc. Figure 3 shows a state diagram generated for the *class clapp* object based on the information of the scenario in figure 2.

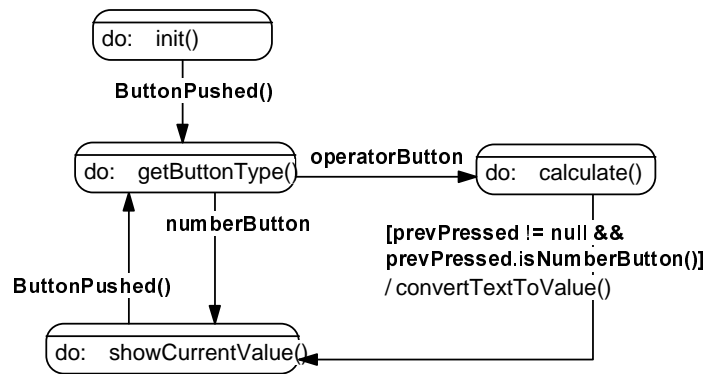


Figure 3. A SCED state diagram generated for the *class clapp* object of a scenario shown in figure 2.

### 3. Instrumenting target system

In order to enable runtime analysis of the object-oriented software, the source code of the target system needs to be instrumented. The instrumentation aims at producing automatically - as a side-effect of running the system - interaction events between interesting objects and conditions that are known to hold. The latter will be transformed into SCED's assertion boxes, while the former are represented as normal event arcs in the scenarios. If the sender and the receiver of an event is the same object, an action block will be generated for the SCED scenario diagram. The resulting sequences of events, i.e. scenarios, will then show the behaviour of the target system in terms of method calls, merged with assertions to describe the selection of branches in various control structures.

Instrumenting the source code is the conventional way of producing run-time information that can be used as the basis of visualization; similar technique is used e.g. in Scene[KoM96]. Also in [SSC96] the animated object communication is shown as a sequence of method calls. However, the architecture-oriented instrumentation in [SSC96] is able to produce a wide range of other kinds of information, viewed using different kinds of diagrams. In [LaN95], the instrumentation is extended to concern also assignment operations, variables, and template functions.

In the system described in this paper both entering and exiting points of a method are recorded as an event; a method call is interpreted as an event sent from the caller to the callee, and the return value as an event sent back from the callee to the caller. If a method doesn't return a value, an unlabeled event from the callee to the caller is sent at the exit point of the method. Constructors are instrumented like any other methods hence enabling the notification of object creations. However, an object creation won't be noticed if it is done by calling default constructors that are not defined in the source code. Destructors could be instrumented similarly. The programming language used in the example discussed in section 4 is Java which has automatic garbage collection. Hence, the instrumentation cannot notice object deletions.

There are basically two approaches for implementing instrumentation described above: instrumenting either method calls or method bodies. Both of these approaches

have advantages and disadvantages. Methods may and usually have several exit points. In the first approach different exit points of a method do not cause any problem, but in the latter approach all the exit points need to be instrumented. Nevertheless, the amount of instrumentation code needed in the latter approach is more likely to be considerably less than in the first approach since there are typically several calling points for a method. Hence, the latter approach is chosen in our experimental system. However, the latter approach provides less means to get various information about the caller than the first one. In our experimental system, we solve this problem in Program Tracer by keeping track of the object that presumably has the control and is thereby the probable caller at a certain moment, that is, the Program Tracer always knows the object that is the receiver of the previous event (the owner of the method previously has been called or the receiver of the last return value). More difficult problem is the instrumentation of return points: a method may return an arbitrary expression, e.g. containing further method calls, making it insufficient to insert the instrumentation code in front of the return statement. In our experiment we have ignored this problem; however, such problems could be fixed automatically by modifying the original program code by eliminating the result expression [KoM96].

Note that the amount of the run-time information showed to the user depends on the instrumentation and tools used for analyzing and showing it. SCED scenario extensions allow us to present method calls and return values, but also assertions implied by if-statements and other control structures.

#### 4. Example: a desk calculator

In this section the run-time behavior of an example system, namely a desk calculator, is animated and visualized following the procedure described in figure 1. This process is run under Windows NT. The source code of the desk calculator is written in Java ([GJS96], [JDK96]); in fact, the desk calculator itself is a simple Java applet. Also the Program Tracer is written in Java. The information transfer from the Program Tracer to SCED is handled via the clipboard using the JDK 1.1 Clipboard API [JDK97].

First, the source code of the desk calculator needs to be instrumented. In our example, this has been made manually. We have not instrumented all method bodies and condition constructs in order to produce a state diagram small enough. In some cases instrumenting both a condition construct and a method body may even produce redundant information, e.g. when a method call is placed inside an if-statement. Automating the instrumentation is discussed in section 6. Since we have chosen to instrument the method bodies instead of method calls, all the exit points need to be instrumented as well, as shown in figure 4. A singlet of the *SCProducer* class is a part of the Program Tracer. Since the Program Tracer knows the caller (the owner of the method previously has been called or the receiver of the last return value), only the callee needs to be told to the Program Tracer besides the actual event name when adding an event to a scenario. The Program Tracer will later consider this callee as a caller of the next event. If the caller and the callee are same, an action box will be inserted to the SCED scenario instead of an event. Lines essential to the Program Tracer are shown as bold text in figure 4.

```
public int getButtonType() {
    Object from = SCProducer.getSingle().getControl();
    SCProducer.getSingle().addScenarioEvent("getButtonType", this);
    if (isNumberButton()) {
        SCProducer.getSingle().addScenarioEvent("numberButton", from);
        return ButtonType.numberButton;
    }
}
```

```

if (isOperatorButton()) {
    SCProducer.getSingle().addScenarioEvent("operatorButton", from);
    return ButtonType.operatorButton;
}
if (isResultButton()) {
    SCProducer.getSingle().addScenarioEvent("resultButton", from);
    return ButtonType.resultButton;
}
SCProducer.getSingle().addScenarioEvent("clearButton", from);
return ButtonType.clearButton;
}

```

Figure 4. Instrumentation of a method body

Figure 5 shows a piece of code instrumented for notifying the Program Tracer not only about the method that has been called ( *calculate()* ) but also about the condition structure that will be shown as an assertion box in SCED scenario. Lines not meaningful for our purposes are replaced with dots (...).

```

private void calculate() {
    SCProducer.getSingle().addScenarioItem("calculate()", this);
    if (prevPressed != null && prevPressed.isNumberButton()) {
        SCProducer.getSingle().addScenarioItem("assertion ~prevPressed != null &&
            prevPressed.isNumberButton()~ for ~" + getClass() + "~\n");
        convertTextToValue();
        ...
    } else {
        SCProducer.getSingle().addScenarioItem("assertion ~NOT (prevPressed != null &&
            prevPressed.isNumberButton())~ for ~" + getClass() + "~\n");
    }
}

```

Figure 5. Instrumentation of a method body and a condition construct

After the instrumentation the source is recompiled and run. Figures 6 and 7 show the final situation of the animation and visualization of the run: figure 6 shows the user interface of the desk calculator applet, and figure 7 the user interface of SCED. There are four overlapping scenario windows shown on the left in the SCED user interface. A state diagram *calcul.sdg* on the right has been synthesized for *class clapp*, which is the actual applet class of the desk calculator and that participates in all these four scenarios. State diagram *calcul.sdg* shows the dynamic behavior of the desk calculator. The last scenario (*calcul4.sc*) describes the calculation "9 / 7 = " whose result is shown by the desk calculator applet.

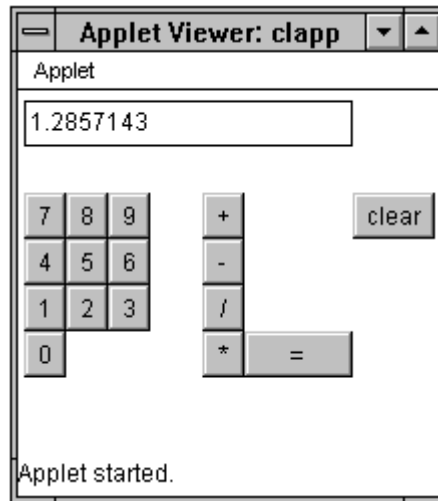


Figure 6. The user interface of the desk calculator applet.

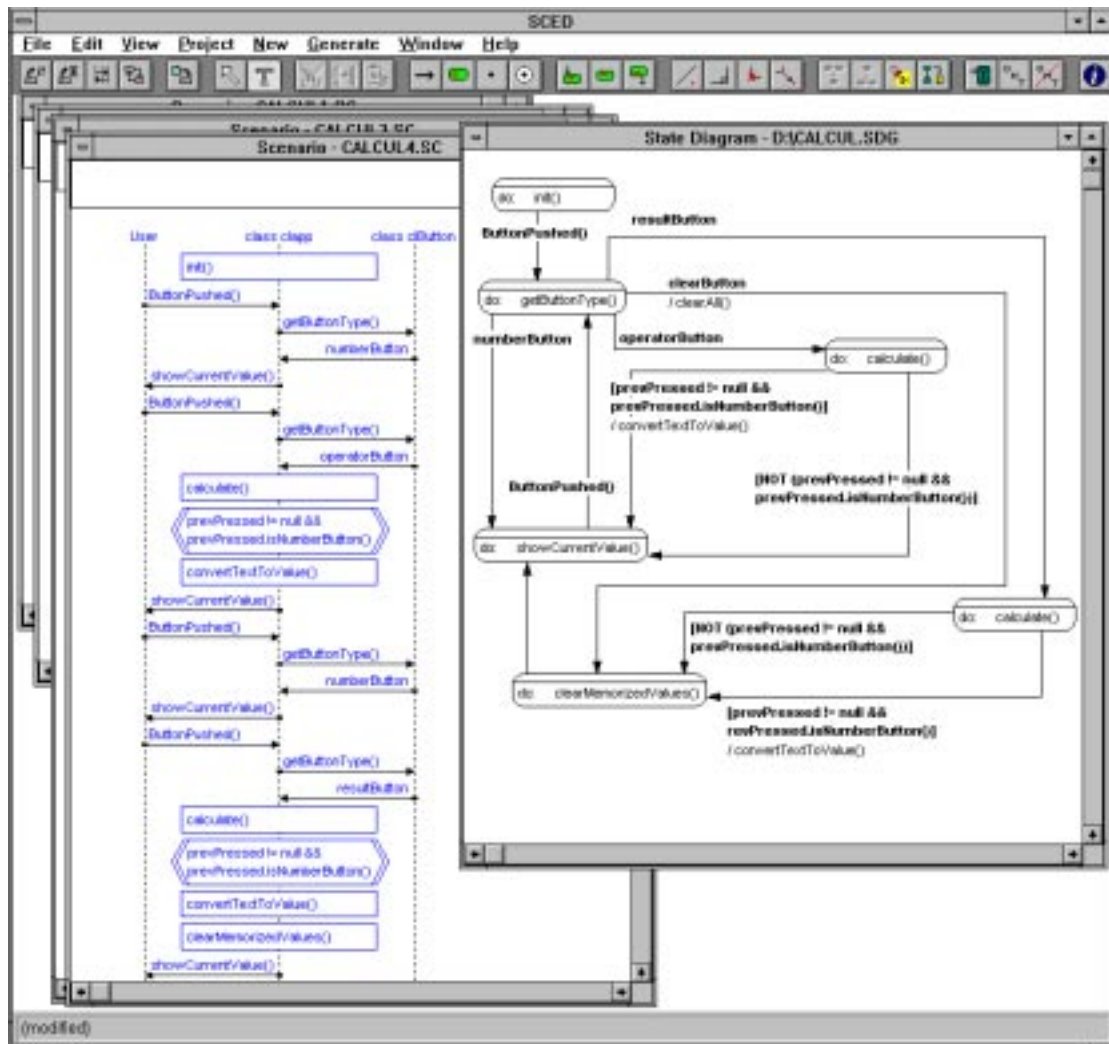


Figure 7. The user interface of SCED. The state diagram *calcul.sdg* has been synthesized for *class clapp* on the basis of four scenarios that result from using the desk calculator.

## 5. Discussion

Extracting the run-time behavior from running systems is important from many perspectives. The benefits gained are considerable, e.g. for reverse engineering, testing and debugging, program understanding, and re-engineering. In this paper we have discussed a technique for generating a state diagram automatically while running and using a target (legacy) system. This technique requires that the source code of the target system is instrumented.

The source code in our desk calculator example has been instrumented manually. However, the instrumentation does not seem to require techniques that could not be automated. One possibility to avoid source code instrumentation would be the usage of a debugger; the contents of the log file produced by the debugger can then be examined and analyzed instead of the source code. The disadvantage of this approach is that the user of such a system would be tied up with a certain debugger. Another approach would be the modification of the Java virtual machine itself, hence allowing the monitoring of the dynamic behavior of any compiled Java program. The great advantage of this is that the source need not be modified or even available. The usability of these possible approaches needs further investigation.

In addition to the instrumentation, the processing of the event trace (i.e. the Program Tracer in our approach) requires further elaboration. Sometimes the user might want to view the behavior of a system on different levels of abstraction through the scenario diagrams. For instance, on the lower level the behavior could be analyzed and showed as a scenario using objects as vertical lines, and on the higher level the vertical lines could represent classes. Raising the analyzing level this way means that the vertical lines of objects of a certain class are "collapsed" to one vertical line only. This kind of abstraction can be called *horizontal abstraction* of a scenario diagram. A higher level view of the behavior could also cause *vertical abstraction*, e.g. by omitting "internal" calls of a single participant in a scenario. Raising the abstraction level also provides means to handle scenario explosion: scenarios tend to grow in both horizontal and vertical directions rapidly when the system to be analyzed gets a bit more complicated. Note that this approach doesn't require any changes to the instrumentation or to SCED. However, a user interface for the Program Tracer is needed for enabling the user to modify the event stream transmitted from the target system to SCED. If inheritance and method overriding is used when constructing Program Tracer in an appropriate way, the abstraction level can be changed dynamically. Some revisions to the current experimental system has already been made using these principles in order to make the Program Tracer part of the system more flexible. In [SSC96] different abstraction levels are offered to the user for visualizing the system dynamics: she can view, e.g. object interaction, class interaction, and communication between frameworks or subsystems. Different diagrams are used for showing different kinds of communication.

## References

- [BiK76] Biermann A.W and Krishnaswamy R.:  
Constructing Programs from Example Computations,  
*IEEE Trans. Softw. Eng.*, **SE-2**, 1976, pp. 141-153.
- [GHJV95] Gamma E., Helm R., Johnson R., and Vlissides J.:  
*Design Patterns: Elements of Object-Oriented Software Architecture*,  
Addison-Wesley, 1995.

- [GJS96] Gosling J., Joy B., Steele G.:  
The Java Language Specification,  
[[http://www.javasoft.com/doc/language\\_specification/](http://www.javasoft.com/doc/language_specification/)],  
Sun Microsystems, August 1996.
- [JDK96] Gosling J., Joy B., Steele G.:  
JDK 1.1.1 New Feature Summary,  
[<http://www.javasoft.com:80/products/jdk/1.1/docs/relnotes/features.html>],  
Sun Microsystems, 1996.
- [FASA94] 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.
- [JDK97]:  
Jawa AWT: Data Transfer,  
[<http://www.javasoft.com/products/jdk/1.1/docs/guide/awt/designspec/datatransfer.html>], Sun Microsystems, February 1997.
- [KoM94] Koskimies K. and Mäkinen E.:  
Automatic Synthesis of State Machines from Trace Diagrams, in *Software Practise & Experience*, **24**, 7, July 1994.
- [KMST96] 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.
- [KoM96] 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)*, Berlin, March 1996, pp. 366-375.
- [LaN95] Lang D. and Nakamura Y.:  
Interactive Visualization of Design Patterns Can Help in Framework Understanding.  
In *ACM SIGPLAN NOTICES*, vol. **30**, no 10, October 1995, pp. 342-357.
- [PKV94] De Pauw W., Kimelman D., and Vlassides J.:  
Modeling Object-Oriented Program Execution, in *Lecture Notes in Computer Science*, vol **821**, eds. M Tokomoro, R. Pareschi, Springer-Verlag, 1994.
- [Rum91] Rumbaugh J., Blaha M., Premerlani W., Eddy F., and Lorenzen W.:  
*Object-Oriented Modeling and Design*, Prentice-Hall, 1991.
- [SSC96] M. Sefika, A. Sane, and R.H. Campbell:  
Architecture-Oriented Visualization, in *ACM SIGPLAN NOTICES*, vol **31**, no 10, October 1996, pp. 389-406.
- [SDV95] Somé S., Dssouli R., Vaucher J.:  
From Scenarios to Automata: Building Specifications from Users Requirements,  
APSEC'95, Brisbane, Australia, 1995.

