A Review and Evaluation of Dynamic Visualisation Tools

Michael J. Pacione

Michael.Pacione@cis.strath.ac.uk

June 2003
(minor amendments October 2003)

Department of Computer and Information Sciences
University of Strathclyde, Glasgow, UK
Abstract

Despite their potential applications in areas including software comprehension, software maintenance, and software reuse, it appears that dynamic visualisation tools are seldom used outside the research laboratory. This report describes a number of dynamic visualisation tools, and evaluates a selection empirically. The role of dynamic visualisation in the software comprehension process and its place in the wider context of software engineering are discussed. Techniques for extracting, analysing, and presenting dynamic information are described. Fourteen dynamic visualisation tools are described and compared in detail. Five tools are then evaluated in the context of a case study using the HotDraw object-oriented framework. The results revealed that the level of abstraction employed by a tool affects its success in different tasks, and that tools were more successful in addressing specific reverse engineering tasks than general software comprehension activities. It was found that no one tool performs well in all tasks, and some tasks were beyond the capabilities of all five tools. This report concludes with suggestions for improving the efficacy of such tools.
Keywords

Abstraction scale, debuggers, dynamic analysis, empirical evaluation, event trace extraction, graph-based representations, interaction diagrams, HotDraw, Message Sequence Charts (MSCs), object-oriented visualisation, program instrumentation, program understanding, reverse engineering, software comprehension tools, Unified Modelling Language (UML).
Acknowledgements

I would like to thank my supervisors, Dr Murray Wood and Dr Marc Roper, for many useful meetings, and for their helpful comments on drafts of this report. I would also like to thank Doug Kirk for giving me the benefit of his experience with JHotDraw and Together, and Dr Wim De Pauw for his comments on Section 3.11.
# Contents

<table>
<thead>
<tr>
<th>Page</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>Title page</td>
</tr>
<tr>
<td>ii</td>
<td>Abstract</td>
</tr>
<tr>
<td>iii</td>
<td>Keywords</td>
</tr>
<tr>
<td>iv</td>
<td>Acknowledgements</td>
</tr>
<tr>
<td>v</td>
<td>Contents</td>
</tr>
<tr>
<td>ix</td>
<td>List of figures</td>
</tr>
<tr>
<td>xiii</td>
<td>List of tables</td>
</tr>
</tbody>
</table>

## Page 1

### 1 Introduction
- 1.1 Report structure
- 1.2 Overview of software comprehension
- 1.3 Static software comprehension techniques
- 1.4 Dynamic software comprehension techniques
- 1.5 Reverse engineering tools support software comprehension
  - 1.5.1 On the relationship between reverse engineering and forward engineering

## Page 4

### 2 Dynamic analysis
- 2.1 Advantages of dynamic analysis for object-oriented systems
  - 2.1.1 On the distinction between dynamic and dynamically-extracted information
- 2.2 Debuggers
- 2.3 Dynamic visualisation tools
- 2.4 Collecting data for dynamic visualisation
- 2.5 Analysing the data produced
- 2.6 Presenting the results
  - 2.6.1 Graph-based representations
  - 2.6.2 UML diagrams
  - 2.6.3 Message sequence charts
- 2.7 Summary

## Page 11

### 3 Dynamic visualisation tools
- 3.1 Characteristics of dynamic visualisation tools
  - 3.1.1 Three criteria for characterising dynamic visualisation tools
  - 3.1.2 A scale to indicate level of abstraction
- 3.2 Program Explorer
  - 3.2.1 Overview
  - 3.2.2 Description
  - 3.2.3 Evaluation
  - 3.2.4 Comparison
  - 3.2.5 Assessment
- 3.3 Scene
  - 3.3.1 Overview
  - 3.3.2 Description
  - 3.3.3 Evaluation
3.3.4 Comparison
3.3.5 Assessment

3.4 Architecture-oriented visualization
3.4.1 Overview
3.4.2 Description
3.4.3 Evaluation
3.4.4 Comparison
3.4.5 Assessment

3.5 ISVis
3.5.1 Overview
3.5.2 Description
3.5.3 Evaluation
3.5.4 Comparison
3.5.5 Assessment

3.6 Dali
3.6.1 Overview
3.6.2 Description
3.6.3 Evaluation
3.6.4 Comparison
3.6.5 Assessment

3.7 Ovation
3.7.1 Overview
3.7.2 Description
3.7.3 Evaluation
3.7.4 Comparison
3.7.5 Assessment

3.8 Reflexion models
3.8.1 Overview
3.8.1.1 AVID
3.8.1.2 RMTool
3.8.2 Description
3.8.2.1 AVID
3.8.2.2 RMTool
3.8.3 Evaluation
3.8.3.1 AVID
3.8.3.2 RMTool
3.8.4 Comparison
3.8.5 Assessment

3.9 Gaudi
3.9.1 Overview
3.9.2 Description
3.9.3 Evaluation
3.9.4 Comparison
3.9.5 Assessment

3.10 Shimba
3.10.1 Overview
3.10.2 Description
3.10.3 Evaluation
3.10.4 Comparison
3.10.5 Assessment
3.11 Jinsight
  3.11.1 Overview
  3.11.2 Description
  3.11.3 Evaluation
  3.11.4 Comparison
  3.11.5 Assessment
3.12 Collaboration Browser
  3.12.1 Overview
  3.12.2 Description
  3.12.3 Evaluation
  3.12.4 Comparison
  3.12.5 Assessment
3.13 Together debugger
  3.13.1 Overview
  3.13.2 Description
  3.13.3 Evaluation
  3.13.4 Comparison
  3.13.5 Assessment
3.14 Together diagrams
  3.14.1 Overview
  3.14.2 Description
  3.14.3 Evaluation
  3.14.4 Comparison
  3.14.5 Assessment
3.15 Tool summary

4 Case study
  4.1 General questions
    4.1.1 Large-scale questions
    4.1.2 Small-scale questions
  4.2 Small-scale questions specified for JHotDraw
  4.3 Together diagrams
    4.3.1 Large-scale questions
    4.3.2 Small-scale questions
    4.3.3 Summary
  4.4 Jinsight
    4.4.1 Large-scale questions
    4.4.2 Small-scale questions
    4.4.3 Summary
  4.5 Reflexion models
    4.5.1 jRMTool
    4.5.2 AVID
    4.5.3 Large-scale questions
    4.5.4 Small-scale questions
    4.5.5 Summary
  4.6 Together debugger
    4.6.1 Large-scale questions
    4.6.2 Small-scale questions
    4.6.3 Summary
  4.7 Case study summary
5 Conclusions and future research
  5.1 Summary and conclusions
  5.2 Contributions of this report
  5.3 Future research directions in dynamic visualisation

References
# List of figures

<table>
<thead>
<tr>
<th>Page</th>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1.1</td>
<td>The relationship between reverse engineering and forward engineering.</td>
</tr>
<tr>
<td>9</td>
<td>2.1</td>
<td>The UML sequence diagram for the Singleton design pattern.</td>
</tr>
<tr>
<td></td>
<td>2.2</td>
<td>The UML collaboration diagram for the Singleton design pattern.</td>
</tr>
<tr>
<td>12</td>
<td>3.1</td>
<td>A scale to indicate level of abstraction.</td>
</tr>
<tr>
<td>44</td>
<td>3.2</td>
<td>The positions of tools on the abstraction scale of Figure 3.1.</td>
</tr>
<tr>
<td>48</td>
<td>4.1</td>
<td>The orrery application. The circles represent astronomical bodies, such as planets and moons, coloured according to their diameter. A blue border around a planet represents atmosphere. The satellite icons represent satellites. The directed arcs indicate gravitational attraction. The toolbar on the left is used to select diagram objects, and to create planets, satellites (orbiting and non-orbiting), atmosphere, and gravity.</td>
</tr>
<tr>
<td>51</td>
<td>4.2</td>
<td>The class diagram generated by Together for the orrery application.</td>
</tr>
<tr>
<td>53</td>
<td>4.3</td>
<td>The sequence diagram generated by Together for the Orbit.MainClass.main() method.</td>
</tr>
<tr>
<td>55</td>
<td>4.4</td>
<td>The sequence diagram generated by Together for the AbstractFigure.moveBy() method.</td>
</tr>
<tr>
<td>56</td>
<td>4.5</td>
<td>The sequence diagram generated by Together for the EllipseFigure.basicMoveBy() method.</td>
</tr>
<tr>
<td>57</td>
<td>4.6</td>
<td>The class diagram generated by Together for the Figure interface.</td>
</tr>
<tr>
<td>58</td>
<td>4.7</td>
<td>The class diagram generated by Together for the Rectangle class.</td>
</tr>
<tr>
<td>59</td>
<td>4.8</td>
<td>The sequence diagram generated by Together for the AbstractFigure.displayBox(Point, Point) method.</td>
</tr>
</tbody>
</table>
The sequence diagram generated by Together for the AbstractFigure.displayBox(Rectangle) method.

The sequence diagram generated by Together for the EllipseFigure.basicDisplayBox() method.

Part of the Jinsight execution view for the orrery application. The coloured horizontal lines represent method calls.

A Jinsight reference pattern view for part of the orrery application.

Part of the Jinsight execution view for the orrery application with repetition detection turned off.

The Jinsight execution view from Figure 4.13 with repetition detection turned on.

Part of the Jinsight object histogram for the orrery application, showing the number of calls to methods of each object. The scale is shown at the top of the window, with black being the lowest and red the highest. Filled rectangles represent objects; outline rectangles represent garbage-collected objects. Diamonds represent the class object of a class.

Part of the Jinsight object histogram for the orrery application, showing the active memory size of the objects.

Part of the Jinsight method histogram for the orrery application, showing the methods called by the selected method.

Part of the Jinsight invocation browser view for the orrery application, showing methods that call the method highlighted in Figure 4.17.

Part of the Jinsight execution view for the second orrery event trace, showing the redraw methods.

Part of the Jinsight call tree for the second orrery event trace, showing the call tree from the invalidate() method.

Part of the Jinsight execution view for the second orrery event trace, showing the implicit control structure of the JHotDraw screen redraw mechanism.
The *Jinsight* call tree showing the implicit control structure of the *JHotDraw* screen redraw mechanism.

Part of the *Jinsight* execution view for the second orrery event trace, with the `EllipseFigure.basicMoveBy()` method selected.

The *Jinsight* call tree for the `EllipseFigure.basicMoveBy()` method.

Part of the *Jinsight* execution view for the second orrery event trace, which shows that `AbstractFigure.displayBox(Point, Point)` calls `MyEllipseFigure.basicDisplayBox(Point, Point)`.

The initial high-level model input to *jRMTool*. Ovals represent high-level system components. Directed arcs represent communication.

The reflexion model computed by *jRMTool*. Ovals represent high-level system components. Directed arcs represent communication; arc annotations indicate frequency. Solid arcs indicate agreement with the analyst’s model (convergences); dashed arcs indicate absences from the analyst’s model (divergences); dotted arcs indicate erroneous communications in the analyst’s model (absences).

The *AVID* cell view at the start of the execution.

The *AVID* cell view partway through the execution.

The *AVID* summary view of the execution.

*Together* debugger output showing the method calls involved in a screen redraw in *JHotDraw*.

*Together* debugger output showing the implicit control structure of the *JHotDraw* redraw mechanism.

The *Together* debugger user interface. The top-left pane shows packages and classes. The top-right pane shows a class diagram. The middle-right pane shows the program code. The bottom pane is the debugger interface, which shows the watches set at the `DEFAULT_MASS`, `MINIMUM_MASS`, `MAXIMUM_MASS`, and `mass` attributes.
The sequence diagram generated by Together for the CH.ifa.draw.standard.SelectionTool.mouseDown() method, illustrating the narrow and deep diagrams produced by dynamic analysis.
List of tables

Page 45       Table 3.1       Tool overview summary
Page 87       Table 4.1       Tools summary comparison
Page 88       Table 4.2       Questions summary comparison
1 Introduction

1.1 Report structure

This report describes the field of dynamic visualisation and its place in the wider context of software engineering. It surveys a number of dynamic visualisation tools and evaluates a selection empirically. This report was motivated by the apparent lack of use of such tools outside the research laboratory. This section provides an overview of software comprehension and reverse engineering in the context of software engineering. The following section describes the role played by dynamic analysis, and the process of dynamic visualisation. Section 3 describes a representative sample of dynamic visualisation tools. Section 4 presents a case study that evaluates a selection of these tools. Section 5 draws conclusions and presents directions for future work in the field of dynamic visualisation. It is concluded that current dynamic visualisation tools do not provide sufficient support for software comprehension when used individually, and suggestions are made for improving the efficacy of such tools.

1.2 Overview of software comprehension

Software comprehension [von Mayrhauser 1995] involves gaining an understanding of the functionality, structure, and behaviour of a software system. Software comprehension has a number of applications in the development and maintenance of software. During the development phase, software comprehension techniques can be used to ensure that the system being developed complies with the system design. During software maintenance, software comprehension can be applied to assist software evolution (extension and contraction of functionality), reengineering (changing existing functionality), and refactoring or restructuring (improving code by making it more extensible or maintainable). Software comprehension also has applications in the field of software reuse [Szyperski 1998, Fayad 1999], where source code or accurate documentation are not always available. Other areas of application include redocumentation (documenting existing software) and legacy system migration (making old systems work in new environments, e.g. the World Wide Web).

1.3 Static software comprehension techniques

Software comprehension techniques can be classified as either static or dynamic. Static techniques analyse the static structure of a system by examining the source or object code of the system under investigation. Static techniques can help in understanding the relationships between classes in a system, and in identifying the system architecture [Müller 1993]. Although software systems written in procedural languages are well suited to analysis with static techniques, aspects of the object-oriented paradigm, such as polymorphism, overloading, and dynamic binding, make it more difficult to gain an understanding of an object-oriented software system using static techniques alone. Gamma et al. [Gamma 1995 pp.22-23] state, “An object-oriented program’s run-time structure often bears little resemblance to its code
structure. The code structure is frozen at compile-time; it consists of classes in fixed inheritance relationships. A program’s run-time structure consists of rapidly changing networks of communicating objects. In fact, the two structures are largely independent. Trying to understand one from the other is like trying to understand the dynamism of living ecosystems from the static taxonomy of plants and animals, and vice versa. […] With such disparity between a program’s run-time and compile-time structures, it’s clear that code won’t reveal everything about how a system will work.” It is for this reason that this report focuses on dynamic visualisation techniques.

1.4 Dynamic software comprehension techniques

Dynamic software comprehension techniques analyse the dynamic behaviour of a software system by extracting information from the system as it is executing. Dynamic techniques can help to illustrate the interactions between objects in a target system, and the flow of control between the system’s components. Dynamic software comprehension techniques address many of the shortcomings of static techniques in the comprehension of object-oriented software systems. A potential disadvantage of dynamic techniques is that they can consider only a subset of the software system’s possible behaviour. While static techniques can analyse the entire system, dynamic techniques analyse only the behaviour evident in the execution trace. It is the responsibility of the analyst to ensure that a suitably representative trace is selected for analysis.

1.5 Reverse engineering tools support software comprehension

Reverse engineering [Cross 1992] describes the process of analysing a software system (complete or incomplete) in order to extract information about its design. Reverse engineering tools exist to support the software developer in his software comprehension tasks. A variety of industrial and academic reverse engineering tools exist, employing either static or dynamic techniques, or an integrated approach. These tools range from relatively simple debuggers, which allow the developer to step through the code execution in a controlled manner and examine variable assignments and method calls as they occur, to interactive visualisation tools, which produce diagrams to the user’s specification, based on a directed analysis of the software system.

1.5.1 On the relationship between reverse engineering and forward engineering

The reverse engineering process can, to an extent, be considered the antithesis of the forward engineering process. The traditional forward software engineering process (linear sequential model, waterfall model, or classic life cycle) comprises: requirements elicitation and analysis, high level design, detailed design, code generation, testing, and maintenance, in that order [Pressman 2000 Sec. 2.4]. The reverse engineering process is equivalent to the reverse of the second, third, and fourth stages of this process. The first stage of reverse engineering is to extract information from the program code. This information can then be analysed to produce
low-level and high-level views of the software system. This relationship is illustrated in Figure 1.1.

**Forward engineering**

- High-level design (e.g. use case diagram)
  - Refine
  - Low-level design (e.g. sequence diagram)
    - Code
    - Source code (e.g. java files)
      - Compile
      - Compiled code (e.g. .class files)

**Reverse engineering**

- High-level recovered design (e.g. use case diagram)
  - Abstract
  - Low-level recovered design (e.g. sequence diagram)
    - Abstract
    - Extracted information (e.g. event trace)
      - Analyse

Figure 1.1. The relationship between reverse engineering and forward engineering.
2 Dynamic analysis

2.1 Advantages of dynamic analysis for object-oriented systems

As described above, dynamic analysis is particularly useful in the context of object-oriented software systems. Dynamic analysis is concerned with the extraction and examination of the dynamic behaviour of a system. The dynamic behaviour describes the actions of a system at runtime; it includes information such as object instantiation and communication, method calls, and branching decisions. In contrast to the collection of static information, dynamic analysis takes place in the context of a running system, rather than by examination of static program code or design documents. The object-oriented programming model often has a complex control flow, with many asynchronous interactions between objects, and points where methods can be called. For example, consider the UML sequence diagram in Figure 4.4, which illustrates the interactions involved in redrawing the screen in a drawing editor. Information for the comprehension of object-oriented systems is hence often difficult to collect and complex to analyse. The large number of object interactions and often unpredictable control flow can result in a large and complicated event trace.

2.1.1 On the distinction between dynamic and dynamically-extracted information

It is important to distinguish between information concerning the runtime behaviour of a software system – dynamic information – and information concerning a software system that is extracted at runtime – dynamically extracted information. Used in the context of this report, dynamic analysis refers to the process of analysing dynamic information about a software system, whether extracted at runtime or otherwise. It should be stressed that dynamic analysis does not supplant static analysis, even in the context of object-oriented systems. However, much of the information traditionally collected by static analysis techniques can also be collected dynamically, thus subsuming much of the functionality of static techniques. For example, that one method calls another method can be revealed through dynamic analysis, but may be also be evident using static analysis techniques, such as a call graph extractor (notwithstanding the difficulties posed to static techniques by object-oriented concepts such as polymorphism, overloading, and dynamic binding, as noted above). As noted above, there are a number of exceptions to the subsumption of static analysis techniques by dynamic analysis in the form of information that cannot necessarily be extracted dynamically. Such information includes, for example, line numbers, comments, and branch conditions. It depends upon the goals of the software comprehension process whether it is more important to know, for example, the conditions pertaining to a branch structure, or whether or not that branch is taken during the execution of the software. Also, as noted above, dynamically extracted information pertains only to the program execution from which it was extracted, and does not necessarily represent all of the possible runtime behaviour of a system. Thus, as in all scientific analysis, the analyst must select the technique appropriate to his task.
2.2 Debuggers

A debugger is a utility that enables the collection of dynamic information from a running system, and has long been part of the software engineer’s tool set. Debuggers operate in an online mode, producing output as the software executes. The software engineer can control the execution of the software by means of the debugger interface, for example by stepping through the code or by suspending and resuming threads of execution. Breakpoints can also be set at points in the code in which the software engineer is interested. Upon encountering a breakpoint during execution, the debugger will output an appropriate message, e.g. ‘Method x called’. The debugger can also be used to examine the values of variables and expressions during execution. A debugger provides a view of a software system at a low level of abstraction (i.e. at a level relatively close to the level of detail provided by the source code itself), and can be invaluable in locating code-level errors a program. However, its low level of abstraction is less useful in software comprehension activities, where a view at a higher level of abstraction (i.e. at a level more distant from that of the code) of the system under analysis is often required [Ball 1996]. An example of output from a debugger is shown in Figure 4.31.

2.3 Dynamic visualisation tools

The foregoing discussion suggests that tools are required to assist the software developer in the collection and analysis of this dynamic information. As with any scientific analysis procedure, dynamic analysis of a software system consists of three phases: collection of dynamic data about the software system; analysis of the data collected; and presentation of the analysis results. In common with many other scientific fields [Nielson 1990], visualisation has been found to be a particularly effective method of presentation for the large and complex data sets produced by dynamic analysis [Roman 1993]. The application of visualisation techniques to the dynamic analysis of software systems is termed dynamic visualisation.

Dynamic visualisation tools typically operate in an offline mode, in which the collection phase precedes the analysis and presentation phases. Walker et al. [Walker 1998] explain that an offline system has two distinct advantages over an online system. Firstly, preprocessing of the entire data set can be carried out prior to the presentation of the results, allowing summary information to be produced for the execution. Such information can be useful in helping the analyst to gain an overall view of the system. Secondly, (a part of) the execution can be analysed repeatedly without requiring the execution to be repeated. This allows the analyst to examine the same execution data in a number of different ways. However, the disadvantage of the offline approach is that it is not possible to explore alternative paths through the execution without rerunning the execution. This makes it difficult to ask “What if…” questions of the system under analysis.

During the data collection phase, dynamic data is extracted from the system under analysis during execution and stored in a repository on disk. The repository can be either a simple file or set of files, or a database. The usual advantages and disadvantages of database systems also apply in this context: while it is quicker to
write a simple text file, a database can be queried more efficiently. The most appropriate repository format depends on the functionality of the visualisation tool.

2.4 Collecting data for dynamic visualisation

The data required for dynamic visualisation of a software system is collected during the execution of that system. This necessitates some form of data collection procedure running either within or alongside the system. One method of collecting this data is by instrumentation of the source or object code of the system. This involves inserting additional statements into the code that generate appropriate output when an ‘interesting’ event occurs during the execution of the system. In the context of object-oriented systems, ‘interesting’ events are usually defined as method calls and returns (when instrumenting the caller) or method entries and exits (when instrumenting the callee). Koskimies and Mössenböck [Koskimies 1996] explain that the advantages of inserting the instrumentation in the caller’s code are that callee methods with multiple return points do not require additional instrumentation, and that information about the caller method is conveniently available. However, the instrumentation of method calls within expressions can appear convoluted. Instrumentation of the source code can also reduce the readability of the code. Code can be instrumented manually or automatically, e.g. using a preprocessor as in Scene [Koskimies 1996]. One method of instrumenting either source or object code is the use of wrappers. Brant et al. [Brant 1998] define wrappers as “mechanisms for introducing new behaviour that is executed before and/or after, and perhaps even in lieu of, an existing method”. Method wrappers were used in Gaudi [Richner 1999] to add instrumentation to the code of the system under analysis.

Another method of collecting the data required for dynamic visualisation is the instrumentation of the environment in which the software system is executing. This method has the advantage that no changes to the source or object code of the system are required. The environment is instrumented to produce appropriate output on the occurrence of relevant events, as with code-level instrumentation. This method is employed in Ovation [De Pauw 1998], where the system under analysis is executed in an instrumented Smalltalk [Goldberg 1983] environment.

An alternative to instrumentation of the code or environment is to run the system under the control of a debugger. Breakpoints set at appropriate points generate the output required. Breakpoints can be set automatically, e.g. at every method entry and exit. This technique is used in Shimba [Systä 2001] to generate trace information for selected methods and control statements. Jinsight [De Pauw 2002] uses a profiling agent to control execution. As with an instrumented environment, running under debugger or profiler control does not require changes to be made to the code.

All of these methods are employed by the tools that are evaluated in Section 3 of this report.
2.5 Analysing the data produced

The huge amount of data produced during the data collection phase must be analysed to produce useful information about the software system. Three ways of reducing the event trace to a manageable size are: selective instrumentation, pattern recognition, and abstraction. These techniques may be used singly or in sequence.

Selective instrumentation involves instrumenting only those methods that are considered ‘important’. An analyst interested in gaining an overall understanding of a system may choose to exclude all methods in library classes (such as java.lang.* and javax.swing.* in Java [Arnold 2000, Gosling 2000, Sun 2003]). Alternatively, an analyst pursuing a specific reverse engineering task, such as investigating how two classes interact, may choose to instrument only the methods of those classes. Selective instrumentation is employed in Shimba.

Pattern recognition is concerned with examining the event trace to detect repetition, in order that this can be factored out to improve comprehensibility. This can performed using string matching algorithms, such as the Boyer-Moore algorithm [Boyer 1977] used in Shimba. Alternatively, Ovation employs a hashing technique to detect and generalise patterns in the event trace.

Abstraction according to specified criteria can be performed on the event trace to raise the level of abstraction from that of individual method calls and returns to some higher level. Gaudi allows trace elements to be clustered arbitrarily into user-defined components to aid understanding of the system under investigation.

Additionally, traces may be split manually or automatically into one or more smaller traces to aid manageability [Systä 2001]. It is also possible to start and stop the instrumentation process, producing trace output only for defined periods of the system’s execution [Walker 1998].

2.6 Presenting the results

The goal of dynamic visualisation is to present dynamic information about the software system under investigation to the analyst in a format that is useful in helping them to achieve their software comprehension tasks. A number of diagramming techniques have been employed in visualisation tools in an effort to achieve this goal; these include graph-based representations, interaction diagrams, statechart diagrams, activity diagrams, and message sequence chart-based representations.

2.6.1 Graph-based representations

Graphs are often used to illustrate the structure or behaviour of a software system. For example, flow graphs are directed graphs that can be used to represent the flow of control in a system; one application of these is in testing [Pressman 2000 Sec. 17.4.1, 17.6.1]. In an object-oriented context, directed or undirected graphs can be used to depict object interactions by representing objects as nodes and messages as directed arcs between nodes. The problem of scalability common to many representations is
particularly evident with graphs – attempting to draw numerous messages between objects can quickly reduce the readability of the diagram. Directed graphs are used in *Program Explorer* [Lange 1995a]. The tool described in [Sefika 1996a] and *Gaudi* employ directed graphs to illustrate interaction between system components, while *Dali* [Kazman 1999] uses undirected graphs to do so. The *reflexion models* used to show this in [Walker 1998] are based on directed graphs. *Shimba* uses undirected graphs to illustrate static method dependencies. An example of a directed graph is shown in Figure 4.30.

### 2.6.2 UML diagrams

Interaction diagrams, statechart diagrams, and activity diagrams are part of the UML diagramming standard [Rumbaugh 1999, OMG 2001], which provides diagrams that illustrate both the static structure and dynamic behaviour of a system. *Interaction diagrams* illustrate interactions, which comprise objects, the relationships between them, and the messages that are passed among them. There are two types of interaction diagrams: *collaboration diagrams* and *interaction sequence diagrams* (*sequence diagrams*). The emphasis of collaboration diagrams is on the structural organisation of the objects, while sequence diagrams emphasise the temporal order of the messages passed between the objects. Though semantically equivalent, the information shown by the two types of diagram differs: collaboration diagrams show the connections between objects, while this is only implied in sequence diagrams. While sequence diagrams show message return values, collaboration diagrams do not.

Figure 2.1 and Figure 2.2 show a pair of corresponding sequence and collaboration diagrams representing the `Singleton.getInstance()` method of the Singleton design pattern [Gamma 1995 pp. 127-134]. This pattern ensures that a class has only one instance in a system, and provides a global point of access to the instance. For example, a system may be connected to a number of printers, but there should be only one print queue. The `getInstance()` method returns this instance. Interaction diagrams solve the issue of scalability inherent in graph-based representations by representing time explicitly along the vertical axis.
Statechart diagrams (statecharts) [Harel 1990] model the behaviour of an individual object as it changes state in response to events. Statecharts emphasise the states in which an object can exist and the transitions between these states. Activity diagrams are flowcharts that describe the flow of control between activities; the Together documentation [TogetherSoft 2001a] describes an activity as “an ongoing, non-atomic execution within a state machine”. While interaction diagrams emphasise the flow of control between objects, activity diagrams emphasise control flow between activities.
Together ControlCenter [TogetherSoft 2001b] synthesises interaction diagrams, statechart diagrams, and activity diagrams from source code. An extended version of UML sequence diagrams and statecharts are used in Shimba. Booch’s object interaction diagram [Booch 1994] (a precursor to the UML sequence diagram) is used in [Sefika 1996a] to illustrate component interaction in a system.

2.6.3 Message sequence charts

Message sequence charts (MSCs) [ITU-T 1996] are similar to UML interaction diagrams. Objects are listed along the top of the diagram, with vertical lines indicating the lifetime of the object. Messages between objects are shown as directed arcs; time progresses downwards. A variation of message sequence charts is used in Ovation [De Pauw 1998]. De Pauw et al. [De Pauw 1998] explain that the tree-structured interaction diagrams – called execution patterns - used in Ovation emphasise the progression of time, rather than the flow of control as in sequence diagrams. They also give the disadvantages of sequence diagrams as being that they do not scale conveniently, and that there are ambiguities in that the ordering of objects on the horizontal axis is arbitrary, and the lifetimes of recursive calls are not easily resolved. The execution patterns of De Pauw et al. also use colour to indicate the class of an object, and label each object with a unique identifier. De Pauw et al. [De Pauw 1998] explain that execution patterns address the perceived shortcomings of sequence diagrams because being unidirectional in both axes makes them more convenient to read, they scale better, and more efficient use is made of space in both axes.

An early form of MSC - an interaction chart - was used in Program Explorer. The OMT [Rumbaugh 1991] event trace diagram (scenario diagram) used in Scene is a variant of the MSC. ISVis [Jerding 1997] used a style of MSC called a Temporal Message Flow Diagram (TMFD) [Citrin 1995]. An example of the execution patterns of De Pauw et al. is shown in Figure 4.11.

2.7 Summary

The section has described the concept of dynamic analysis and its advantages for object-oriented systems. It has also described the process of dynamic visualisation, comprising collection, analysis and presentation of trace data. The relative merits of the three main diagram techniques were discussed. The following section examines a number of dynamic visualisation tools in detail and compares their functionalities.
3 Dynamic visualisation tools

3.1 Characteristics of dynamic visualisation tools

3.1.1 Three criteria for characterising dynamic visualisation tools

From the foregoing discussion, three distinguishing criteria regarding dynamic visualisation tools can be identified. The first of these is the method used to extract the dynamic information from the software system. Techniques include instrumentation of the source or object code (e.g. using wrappers) or environment, or running the system under the control of a debugger or profiler.

The second criterion is the methods of analysis that are applied to the extracted data to improve its comprehensibility and usefulness to the analyst. These include selective instrumentation, pattern recognition, abstraction, trace splitting, and suspension/resumption of tracing.

The third criterion is the method by which the results of the visualisation are presented to the analyst. Diagramming techniques are typically based on graphs, UML diagrams, or message sequence charts.

3.1.2 A scale to indicate level of abstraction

The combination of these three criteria determines the level of abstraction at which the dynamic visualisation tool operates. This report proposes a scale for the classification of dynamic analysis tools based on their level of abstraction; this is illustrated in Figure 3.1. An ordinal scale is used to assign a value (or range of values) from one to five to a tool, based on its position relative to the five indicated reference points. At the microscopic end of the scale, debuggers (1) are representative of the lowest level of abstraction that a dynamic analysis tool can produce. At the opposite, macroscopic, end are tools that provide a broad overview of an entire software system at a high level of abstraction (5). The middle portion of the scale ascends from tools that illustrate method calls and returns (2), through tools giving an object- or class-level representation of the system (3), to tools that provide an architectural-level view of the system (4). The program code itself can be considered to be at level 0. The application of this scale is not restricted to the assessment of dynamic visualisation tools. It is equally applicable to diagrams, and indeed other forms of documentation, at any stage of the software engineering life cycle.
The remainder of this section examines a representative sample of the dynamic visualisation tools that have been produced. Each tool is assessed according to the three criteria above, and placed on the abstraction scale described above.

3.2 Program Explorer

3.2.1 Overview

References: [Lange 1995a, Lange 1995b, Lange 1997]

Availability: Not available

Level of abstraction: 2

Analysis language: C++

Implementation language: C++

Platform: IBM AIX

3.2.2 Description

Lange and Nakamura [Lange 1995a] discuss the investigation of object-oriented frameworks by means of visualisation that focuses on identifying design patterns. The paper describes Program Explorer, a tool that uses a combination of static and dynamic information to visualise C++ programs. The program has a GUI front end, and queries are formulated in Prolog. Static and dynamic information are stored in a single “Program Database”. The static information for this database is gleaned from files output by a compiler. The system consists of a C++ program database; an “instrumentation utility” that instruments the C++ source code; a “Trace Recorder” that is linked with the program under analysis to capture the event trace during execution; and “Program Explorer”, which is used to control the execution of the program, and present the static and dynamic information using its GUI.
The tool presents the visualisations using a graph-based representation and interaction charts. These visualisations can be navigated step-by-step in a hypertext-like manner (e.g. expand a node in the graph, explore a relationship between two nodes). The dynamic information is extracted by automatic instrumentation of the source code. Further information on the instrumentation technique is available in [Lange 1995b]. A version that uses runtime trapping instead of source code instrumentation, thus eliminating the need for an extra compilation stage at the expense of execution speed, is discussed in [Lange 1997].

The visualisation can be localised by allowing the user to set breakpoints at a variety of points in the source code, including classes, objects, function calls, etc. This also allows the user to switch the tracing on and off, limiting the size of the trace. It does not appear that any automatic analyses (e.g. behavioural pattern matching as in Shimba) are applied to limit the size of the event trace and, hence, the resultant diagrams.

Lange and Nakamura [Lange 1995a] explain how identifying design patterns can help in framework understanding, using examples from the Interviews framework [Linton 1992]. It is argued that patterns can help in two ways. Firstly that, once identified in the software comprehension process, they can help to “fill in the blanks” about the rest of the system. Secondly, patterns can provide a starting point for the exploration of a system. Lange and Nakamura [Lange 1995a] comment that although some automation using heuristics may be possible, it is unlikely that the identification of patterns in the visualisation could be fully automated. The system relies on the user being “pattern-literate”, and being able to identify the semantics of design patterns from the method names and interactions between objects.

### 3.2.3 Evaluation

Lange and Nakamura [Lange 1995a] cite user reports that the tool was useful for three types of task, namely: in supporting the understanding of certain specific C++ frameworks; for reviewing designs, allowing visualisation of the implemented design in comparison to the original design; and for visually debugging the application logic of C++ systems. Lange and Nakamura [Lange 1995a] state that Program Explorer provides framework developers with “abstract” design pattern views, and “microscopic” views that provide sufficient detail as to make source code superfluous in the software comprehension process.

Lange and Nakamura [Lange 1995a] explain that Program Explorer’s ability to handle complex frameworks such as Interviews and CommonPoint [Taligent 1994] is attributed mainly to: the integration of static and dynamic information; ease of view navigation and interaction; and the ability to trace selectively; and user control over the execution.

### 3.2.4 Comparison

At the time of writing [Lange 1995a], there do not appear to be any tools with functionality comparable to that of Program Explorer. Lange and Nakamura [Lange
discuss briefly two static analysis tools - CIA++ [Grass 1992] and GraphLog [Consens 1993] - and two dynamic analysis tools - Object Visualizer [De Pauw 1993, De Pauw 1994] and HotWire [Laffra 1994]. Both dynamic tools are based on the same instrumentation mechanism, which is less accurate than that used in Program Explorer in that it lacks information on implicit functions, variable usage and variable values. Object Visualizer is an object-oriented profiling tool, while HotWire is a visual C++ debugger. Lange and Nakamura [Lange 1995a] explain that it is HotWire that is most similar to Program Explorer. Both HotWire and Program Explorer generate microscopic visualisations concerning the state and behaviour of objects, while Object Visualizer provides a more general overview, similar to a profiling tool.

Jerding and Rugaber [Jerding 1997] observe that Program Explorer is not intended to give an overall understanding of a software system, but to focus on specific classes or objects. The analyst must be aware of what he is looking for, or where in the execution it occurs, before he begins his analysis.

Walker at al. [Walker 1998] note that the analyst must possess an in-depth knowledge about the software system under analysis in order to query usefully the fine-grained models produced by Program Explorer.

De Pauw at al. [De Pauw 1998] argue that Program Explorer bridges the gap between microscopic and macroscopic extremes by its use of interaction diagrams. However, they note that such diagrams are inconvenient and suffer from the difficulties cited in Section 2.6.3.

Richner and Ducasse [Richner 1999] note that the highest level of abstraction supported by Program Explorer is the class or object level.

Systä et al. [Systä 2001] observe that the analyst cannot specify how the event trace is split into sequence diagrams, and that the level of abstraction of these diagrams is fixed.

3.2.5 Assessment

The comments made in [Jerding 1997] suggest that Program Explorer is more suited to specific reverse engineering tasks than overall software comprehension, as the analysis must be focussed clearly. Therefore, it would be expected that Program Explorer might struggle with general software comprehension tasks, but could perform well in specific reverse engineering tasks, depending on the level of detail required for useful analysis.

3.3 Scene

3.3.1 Overview

References: [Koskimies 1995, Koskimies 1996]

Availability: ftp://ftp.ssw.uni-linz.ac.at/pub/Oberon/Scene
3.3.2 Description

Koskimies and Mössenböck [Koskimies 1996] describe a tool called Scene (Scenario Environment) that produces scenario diagrams from a dynamic event trace. Calls can be expanded and collapsed to simplify the scenario diagram. A hypertext approach enables the analyst to click various areas of the diagram (e.g. a method call or an object) to jump to a related document (e.g. a point in the source code or a class interface). An externally-produced class diagram can also be linked to the scenario diagram. The scenario diagrams are partitioned to display only as many objects as fill the screen horizontally, thus eliminating horizontal scrolling. Calls to ‘uninteresting’ objects can be filtered out in the diagram by selecting the object(s) to retain or discard. Calls can be expanded and viewed in a ‘single-step mode’ where subsequent events are displayed one by one in a separate window. For any call event (or for the whole diagram), a call summary can be viewed in the form of a call matrix.

The system is implemented in the Oberon-2 language [Mössenböck 1991] and runtime environment [Reiser 1991, Wirth 1992], and traces programs in this language. Oberon-2 is a hybrid language, which, in addition to the object-oriented concepts of classes and methods, also supports modules and procedures. The trace is obtained by automatically instrumenting the source code using a preprocessor, which is then compiled and executed. The event trace is then input to Scene, which produces a scenario diagram. ‘Uninteresting’ modules (e.g. those related to mouse events in a GUI) can be excluded from the instrumentation, or instrumented manually by the analyst at their discretion.

A problem identified in [Koskimies 1996] was the lack of support for understanding of the relationships between multiple scenario windows, which represent a hierarchy. Future work includes automatic production of object state information, instrumentation of object rather than source code, and application of Scene to other languages such as C++. More detail on Scene is given in [Koskimies 1995a].

3.3.3 Evaluation

Koskimies and Mössenböck [Koskimies 1996] report that Scene has been used to analyse a number of “framework-like” systems, including a compiler construction framework [Koskimies 1995b] and a graphics editor [Templ 1994]. They believe that Scene is most beneficial in the analysis of frameworks, as understanding their complex dynamic behaviour is vital for reuse.
3.3.4 Comparison

The scenario diagrams used in Scene are similar to Program Explorer's interaction charts, and represent the same level of abstraction. Both tools extract dynamic information through automatic instrumentation of the program source code.

3.3.5 Assessment

The similarity of the representations in both Scene and Program Explorer would suggest that Scene may also be better suited to targeted reverse engineering tasks than overall software comprehension activities. However, the diagram manipulations and summary generation supported by Scene would be expected to give it an advantage over Program Explorer in overall comprehension tasks.

3.4 Architecture-oriented visualization

3.4.1 Overview

References: [Sefika 1996a]
Availability: Not available
Level of abstraction: 4
Analysis language: C++
Implementation language: Berkeley YACC, SWI Prolog, Tcl
Platform: UNIX

3.4.2 Description

Sefika et al. [Sefika 1996a] discuss the concept of architecture-oriented visualization, which is concerned with the visualisation of architecture-level components of software systems, e.g. subsystems, frameworks, design patterns. Sefika et al. [Sefika 1996a] state that it is often architectural-level questions that are most useful in understanding software systems, but that answering such questions using traditional programming tools is difficult for a number of reasons. Firstly, the volume of data generated by “flat” instrumentation of method calls and returns is too great for architectural-level understanding, and its collection disrupts the system under analysis. Secondly, the more abstract architectural structures are hidden from instrumentation. Thirdly, prior systems have scant support for multiple perspectives or hierarchical navigation, making it difficult to analyse the information from various abstraction levels and design aspects that is required to discern the software architecture.

The user has a choice of diagrams for different purposes. In the case studies, a bar chart is used to display the number of processes blocked per subsystem; space-filling diagrams to illustrate process blocking statistics at sub-framework, inheritance structure and class levels, and for particular class instances; ternary diagrams to show communication between sub-frameworks and subsystems; affinity diagrams to show communication between classes of a sub-framework and classes of a subsystem; and object interaction diagrams to show object interactions. The system supports multiple simultaneous diagrams of combined static and dynamic information, with hyperlinks between diagrams. The tool described uses an online approach.

The two principal constraints on the design of the system were that it must incur low spatial and temporal overheads, and that it must be flexible enough to allow the analyst to change the data extraction technique conveniently. The key architectural design decisions were identified as: how the query interpreter maps architectural units to instruments; how the instrument managers control instruments; how data collectors visit instruments to obtain data; and how events should be directed to interested instruments. The structure of the system is based around events being received by an event sensor and passed to an event announcer, which informs an instrument at the relevant level of abstraction (i.e. that selected by the user) that the event has occurred. The information is obtained from method-level instrumentation contained in the Choices operating system. Queries are formed automatically via the GUI, or can be entered manually ([Sefika 1996a] gives the grammar in Extended BNF notation [Aho 1986]).

At the time of writing, this appears to be the first paper to consider dynamic architecture-oriented visualization. Potential for future research is identified in combining the system with a code refactory to automate design repair; in utilising the instrumentation techniques in an optimising compiler; and integration of the system with everyday programming tools such as debuggers and code browsers. More generally, Sefika et al. [Sefika 1996a] expect the pervasive instrumentation technique to provide analysts with better user interfaces and views, particularly 3D views exploiting virtual reality technology. Unfortunately, this latter development does not appear to have materialised.

3.4.3 Evaluation

Sefika et al. [Sefika 1996a] present two case studies based on the Choices object-oriented operating system [Campbell 1993], written in C++. One is related to identifying a system bottleneck, and the other related to analysing subsystem cohesion and coupling.
The performance of the architecture-oriented tool is compared to that of traditional “flat” method-level instrumentation tools, and the improvements of architecture-oriented visualization over traditional visualization are identified as follows. Firstly, architecture-oriented instrumentation utilises knowledge of the software structure, enabling it to reduce the volume of data that must be collected, lowering the overhead of dynamic analysis. This reduction also decreases the volume of data sent to the visualiser, and hence also reduces the requirement for analysis to improve the comprehensibility of the data. The volume of data collected is further reduced in the architecture-oriented approach as event sensors and instruments are enabled according to the requirements of the query. The hierarchical organisation of the instruments allows information about system structure to be obtained quickly.

In terms of data generation, a graph comparing architecture-oriented instrumentation with traditional instrumentation reveals that architecture-oriented instrumentation dramatically reduces the trace size as the level of abstraction employed increases. In terms of instrumentation overhead, two graphs illustrate clearly that architecture-oriented instrumentation reduces the analysis overhead, as the volume of data and time required for collection are decreased, and instruments are enabled depending on the components of the current query.

Sefika et al. [Sefika 1996a] note that architecture-oriented instrumentation is not entirely without cost. While traditional instrumentation increased the size of the executable by 7.8%, architecture-oriented instrumentation caused an increase in size of 14.3% in the worst case. This difference is due to the addition of code for query processing and instrument management required by architecture-oriented instrumentation. Sefika et al. [Sefika 1996a] feel that this is an acceptable trade-off, given the benefits of architecture-oriented instrumentation and falling memory prices, and the fact that most of the architecture-oriented instruments in the code will be unused until explicitly required.

3.4.4 Comparison

Jerding and Rugaber [Jerding 1997] note that the goals of [Sefika 1996a] are similar to those of ISVis in visualising a system from a variety of architectural levels. However, they point out that some of the views described in [Sefika 1996a] are tightly coupled to the subject system domain, rather than being generally applicable to software architectures. They speculate that this could be because the tool was applied only to an operating system.

Walker et al. [Walker 1998] state that the higher-level visualisations provided by the [Sefika 1996a] tool are an improvement over earlier techniques in analysing component interactions in large systems. However, Walker et al. [Walker 1998] also explain that the tool is not as flexible as it could be. While an online approach provides a connection between system execution speed and the speed shown in the visualisation, this places restrictions on the analyses that can be carried out, as discussed in Section 2.3. The approach taken in [Sefika 1996a] of using predefined abstraction types built into the tool, while gathering dynamic information effectively, reduces the flexibility of the technique by making it more difficult for the analyst to adapt it to a different system. The reflexion model technique described in [Walker
1998] can conveniently be applied to a variety of systems, partly due to the
decoupling of the data collection and visualisation components.

Richner and Ducasse [Richner 1999] note that while the [Sefika 1996a] tool is one of
the few tools to support architectural-level visualisations, the approach taken requires
application-specific instrumentation, unlike Gaudi.

Systä et al. [Systä 2001] observe that the [Sefika 1996a] tool requires the analyst to
select the abstraction level and views to be produced before running the software
system to be analysed. Shimba is more flexible: it does not have this requirement, and
provides a variety of techniques to allow the analyst to construct abstractions from the
low-level views produced.

3.4.5 Assessment

The architectural-level visualisations produced by the [Sefika 1996a] tool suggest that
it would perform well in general software comprehension tasks. If appropriate views
and abstraction level were selected, it could also be useful in specific reverse
engineering tasks. However, the evaluation in [Sefika 1996a] was in the context of an
operating system only, and it remains to be seen whether the technique will perform
well when visualising other types of application.

3.5 ISVis

3.5.1 Overview

References: [Jerding 1997]
Availability: http://www.cc.gatech.edu/morale/tools/isvis/isvis.html
Level of abstraction: 4
Analysis language: C, C++
Implementation language: C++
Platform: Sun Solaris

3.5.2 Description

Jerding and Rugaber [Jerding 1997] describe a tool called ISVis (Interaction Scenario
Visualiser) for identifying software system architecture. Static information is extracted
from files generated by the Solaris compiler. An instrumentor then combines this
static information, the source code, and information from the analyst about what to
instrument, and generates instrumented source code. This is then compiled, and then
executed according to the desired usage pattern, and event traces are produced. The
ISVis trace analyser then converts this information into a set of scenarios and
involved *actors* that are stored in a program model. The user then queries views of this program model. *ISVis* has a Main View and a Scenario View. The Main View lists the actors in the program model, including user-defined components, files, classes, and functions, and the scenarios and interactions in the program model. A Scenario View can be opened for any scenario in the model, which takes the form of a Temporal Message Flow Diagram (also called TMFD, interaction diagram, message sequence chart, event-trace diagram). A global overview is shown using an Information Mural [Jerding 1995]; this allows the analyst to visually identify repeated patterns in the execution. An option allows actors to be grouped by containing file, class, or component actors. Another option allows the user to select an interaction or class of interactions and define them as a scenario, which can then be abstracted out and replaced in the diagram by a reference to the scenario. Interaction patterns can also be identified by a technique similar to regular expression matching. Jerding and Rugaber [Jerding 1997] compare the interaction patterns of *ISVis* to design patterns [Coplien 1995, Gamma 1995], stating that interaction patterns are a result of the implementation of design patterns, and constitute low-level evidence of their existence.

The relationship between the two views and the program model is an implementation of the Observer design pattern [Gamma 1995 pp. 293-303], and an example of the Model-View-Controller (MVC) architecture used in languages such as Smalltalk [Krasner 1988]. The Observer design pattern defines a one-to-many relationship between objects, such that when one object changes state all its dependent objects are notified and updated automatically. For example, objects representing different views of the same data, e.g. a pie chart, a bar chart, and a spreadsheet, could be registered to observe the data source and hence be updated automatically when the data source changed. The Observer pattern allows consistency between cooperating objects, without making them tightly coupled which would reduce their reusability. *ISVis* allows the analyst to save the event traces and program model for future analysis. The process of reading in the trace, creating the program model, creating scenarios and architectural models, and viewing the results is iterative, with each analysis building on the results from the previous analysis. Analysis sessions can be loaded and saved. *ISVis* can simultaneously analyse a number of traces from one system.

Jerding and Rugaber [Jerding 1997] suggest future applications of *ISVis* as more effectively suggesting patterns to the analyst, and import/export of components from/to other tools. Future work is to include interoperation of *ISVis* with the *Balboa* machine-learning finite state machine generation tool [Cook 1995], and with the *SAAMTool* architectural analysis tool [Kazman 1994].

### 3.5.3 Evaluation

*ISVis* is applied to a case study involving adding functionality to the Mosaic web browser [NCSA 2000]. The authors term the problem of finding where in a system to insert an enhancement “architectural localization”. The high-level process of architectural localization during the case study consisted of producing scenarios, removing interactions that do not pertain to the functionality being localised, using the information mural to browse the scenarios and identify patterns, using pattern
matching to find scenarios similar to those already identified, then relating this behaviour to the source code.

The principal strength of ISVis was reported to be its support of the abstraction process by means of interaction patterns. This frees the analyst from the compute-intensive work and allows them to identify semantically those patterns that are relevant to the task in hand. This allows the analyst to manually perform inferences that would not be considered by a wholly automated approach. The authors emphasise the importance of appropriate usage scenarios being chosen, as these have a direct effect on the analyst’s ability to identify patterns. The problem of selecting a suitably representative trace is a key concept in dynamic analysis, as discussed above in Section 1.4 and Section 2.1.1.

A weakness is given as the complexity of the user interface, which is attributed to its rich features. The importance of scalability is emphasised, as architectural visualisation is only useful if the system is large enough to benefit from such analysis. It is reported that the information mural was effective at compressing the large volume of data.

3.5.4 Comparison

Unlike Ovation [De Pauw 1998] and Jinsight [De Pauw 2002], ISVis does not automatically identify patterns of repeated execution; ISVis requires the analyst to identify such patterns.

Richner and Ducasse [Richner 1999] believe that Gaudi complements ISVis in that while both tools acknowledge that higher-level views are required for architectural understanding, ISVis concentrates on pattern detection, while Gaudi allows the analyst to specify the type of view used.

Systä et al. [Systä 2001] observe that source files are the lowest level of granularity that can be excluded from the trace in ISVis, while Shimba allows individual classes and methods to be excluded. Shimba also allows more flexible construction of abstractions. However, Shimba only allows pattern searching using exact string matches, and patterns must be contained within a single sequence diagram.

3.5.5 Assessment

As with the [Sefika 1996a] tool, the architectural-level visualisations used in ISVis would appear to lend themselves well to general software comprehension. The tool may also be useful in specific reverse engineering tasks, depending on the level of abstraction required for the task.
3.6 Dali

3.6.1 Overview


Availability: http://www.sei.cmu.edu/ata/products_services/dali.html

Level of abstraction: 4

Analysis language: Retargetable

Implementation language: Perl, Tcl/Tk

Platform: Linux, Sun Solaris

3.6.2 Description

Kazman and Carrière [Kazman 1998, Kazman 1999] describe a tool called the Dali Workbench, which is designed to help with the extraction of program architecture. It is designed as a lightweight, flexible tool that integrates other tools, the argument being that no single tool is adequate for architectural extraction. Kazman and Carrière [Kazman 1999] argue that software architecture is a “shared hallucination” – it exists from the various points of view of people involved with the software. It is thus argued further that a human element is essential in the process of architectural extraction. The goal of Dali is to assist the analyst in the analysis of software architecture. This implies a need for the reconstruction of architectural representations of the system. Kazman and Carrière [Kazman 1999] list the main contributions of Dali as its use of a central data repository to integrate system information; its use of a common language (SQL) to enable the combination of views and user-defined pattern matching; and its assessment of such patterns as a metric for architectural conformance.

Four iteratively applied techniques are involved in the process of reconstructing software architecture using Dali. Firstly, static information from source artefacts, such as the program code, and dynamic information from the output of profilers or coverage tools is used to create extracted views of the system. These views represent the implemented architecture of the system. Secondly, the extracted views are combined to produce fused views giving a more complete representation of the architecture. Thirdly, the analyst defines a number of architectural patterns that represent his understanding of the implemented architecture, which are used to create refined views. Fourthly, the refined views are visualised to allow the analyst to compare the implemented architecture to the designed architecture.

The extraction component of Dali extracts information using tools such as lexical analysis, parsing, and profiling tools, then combines this information. This information is stored in a central repository (a relational database). The contents of the repository can be visualised and manipulated, and analyses can be performed on them. The various tools that are used with the Dali Workbench are not fixed in its specification, but examples include the following tools: Lightweight Source Model
Extraction (LSME) [Murphy 1996] for extraction of static information; gprof for extraction of dynamic information; PostgreSQL (based on POSTGRES [Stonebraker 1990]) as the relational database; SQL for view fusion and architectural pattern definition; and RMTool [Murphy 1995] for analysis.

Kazman and Carrière [Kazman 1999] state that the intention was not to provide an ultimate solution, but to develop an extensible environment for tool integration. Future research includes extending the scope of Dali to use other languages and larger systems (e.g. legacy COBOL systems) – at the time of writing, Dali had been used on systems up to 200 KLOC (thousand lines of code) in C, C++, Objective C, and Fortran; there is evidence of such extension in [O’Brien 2002]. There is also the possibility of integrating other tools, such as to enable the import and export of architecture representations in ACME [Garlan 1997] or UniCon [Shaw 1995]. It is also hoped to improve user interaction, with the addition of a history/undo feature in the short term, and the ability for the user to directly manipulate the architecture and have the system infer appropriate architectural rules. Finally, Dali could be used to guide architectural evolution, e.g. in determining how difficult it would be to change the connection mechanisms of an architecture; this could be useful in web-enabling legacy systems or distributing them via CORBA.

3.6.3 Evaluation

Kazman and Carrière [Kazman 1999] describe the application of Dali to two C++ systems: VANISH [Kazman 1996], which has a well-designed architecture, and UCMEdit [Buhr 1996], which has no designed architecture. The study describes the stages of extracting the information, forming “fused views”, then applying patterns (expressed as SQL queries) to simplify the resultant visualisation (application-independent patterns, common application patterns, then application-specific patterns). The analyst carrying out the architectural extraction would appear to have to be either a very good software engineer, or even to be intimately familiar with the system under investigation. The sorts of manipulation that are carried out involve e.g. the grouping of methods and variables into their associated classes, and the grouping of functions and header files into their associated classes. The case studies extracted the as-implemented architecture from both systems, but, as would be expected, found the VANISH architecture much more useful. The analysts were also able to identify some architectural exceptions and points for improvement in the VANISH architecture using the extracted model. [Kazman 1999] notes that a good architecture is characterised by functional consistency.

O’Brien [O’Brien 2002] describes three case studies in which Dali was employed in an industrial architecture reconstruction project at Nokia. The system involved in the first case study was a network management system consisting of 500 KLOC of C; the goal was to understand how the system could be improved. The second case study concerned another network management system consisting of 100 KLOC of Java; the goal was to understand the system and determine whether it could be reused. The third case study involved a mobile phone system consisting of 1 MLOC (million lines of code) of C++; the goals were to examine the way in which this application was integrated with the operating system, and to determine whether a specific component could be extracted and reused. O’Brien reports that the architecture reconstruction
efforts were successful in each of these contexts with their various goals, and that the architects found the Dali views to be useful. However, a difficulty was identified concerning the static analysis of the C and C++ systems. It was found that identifier names extracted from the source code were often not unique, and could not be discriminated between without compiling and linking. O’Brien concludes that architecture reconstruction requires tool support, and that such tools are available. However, research is required to improve the reconstruction process and the tools that support it.

3.6.4 Comparison

Systä et al. [Systä 2001] observe that Dali uses a single merged view to represent both static and dynamic information about the software system, whereas Shimba uses separate, linked views to separate static and dynamic information.

3.6.5 Assessment

As with the other architecture-level visualisation tools in this report, Dali would appear to be well-suited to general software comprehension tasks. The intended role of Dali as an architectural extractor may make it less suitable for specific reverse engineering tasks. However, performance in either task will depend on the ease with which appropriate architectural patterns can be identified and useful architectural views built.

3.7 Ovation

3.7.1 Overview

Availability: Not available
Level of abstraction: 2
Analysis language: C++, Java, Smalltalk
Implementation language: C++
Platform: IBM AIX

3.7.2 Description

De Pauw et al. [De Pauw 1998] describe a tool for visualising programs using an execution pattern view, which is a variation of Jacobson’s interaction diagrams [Jacobson 1992]. The technique is based on that used in Ovation [De Pauw 1993, De Pauw 1994], and has since been implemented in Jinsight [De Pauw 2002]. De Pauw et
al. recognise the inherent information overload problem, noting that both statically complex and small, repetitive programs can produce huge traces. They state [De Pauw 1998] that dynamic execution trace data can be comprehended if it is summarised into distinct, abstract portions and detail is provided to the analyst on demand, and if patterns in the trace can be detected and generalised. The execution pattern view achieves these requirements by allowing the analyst to examine program execution at various levels of detail, with information supplied only on demand, and by extracting and visualising generalised patterns in the trace. Ovation can visualise C++ or Java programs using traces generated from the VisualAge development environment [IBM 2003a], and Smalltalk programs through instrumentation added to the Little Smalltalk [Budd 1987] and VisualAge Smalltalk [IBM 2003b] environments.

De Pauw et al. [De Pauw 1998] observe that, while interaction diagrams are an improvement on directed graphs for illustrating program interactions, they do not scale up well to larger execution traces. The execution pattern view instead uses a tree structure, emphasising the progression of time, rather than control structure. Colour is used to indicate the class of an object, and a unique object ID appears in each object box. In the execution pattern views, horizontal space is mapped to the call sequence, not the object population, and vertical space is also used more efficiently. The view can be explored by searching for execution patterns based on a number of criteria, such as the involvement of a specific class, object, or method. Subtrees can be collapsed and expanded, allowing the user to “drill down” to focus on interactions of interest while excluding extraneous detail. The context of the view can also be changed by moving up or down the call hierarchy. Filtered expansion is also possible, for example by expanding only those nodes in the tree that lead to a certain type of object. The system can automatically detect repetition, either in the form of iteration (shown vertically) or recursion (shown horizontally). Zooming and panning the view is also supported. Flattening can be used to limit the horizontal depth by collapsing only the receiver of the message. Underlaying saves horizontal space by hiding all the messages sent by the underlaying class and displaying call recipient objects on top of the object that initiated the call. These techniques allow the analyst to navigate the execution one step at a time. A number of alternative charts for representing subtrees are available, including class legends, and class communication graphs. Other possible charts could include a CPU time meter, a call matrix [De Pauw 1993], or an instance histogram [De Pauw 1993]. To aid comprehension, “flyovers” and zooming (without scaling method names) are supported.

Ovation supports generalized (i.e. non-identical occurrences) pattern matching for detecting patterns of similar execution. The generalization criteria for pattern matching implemented in Ovation are those that De Pauw et al. report [De Pauw 1998] that programmers found most useful: object identity, class identity, message structure, depth-limiting, repetition, polymorphism, associativity, and commutativity. To implement this generalisation, the tool assigns a hash value to each subtree of the execution tree. The subtree hash code is formed from the hash codes of the subtree’s children and values in the subtree’s root. The values used to form the hash code depend on the matching criterion specified, e.g. method names (method and class names would be used) or class names (class names would be used). The hash values are stored in a pattern dictionary, which records summary statistics for each entry (e.g. frequency of this pattern). De Pauw et al. [De Pauw 1998] argue that the execution pattern view bridges the gap between microscopic and macroscopic
visualisation representations by providing a view of the entire trace, with more detail available on demand.

De Pauw et al. [De Pauw 1998] conclude that execution patterns have three key benefits for object-oriented visualisation. Firstly, they provide a convenient representation of object-oriented communication. Secondly, similar execution patterns can be generalised. Thirdly, execution patterns can help in the assessment of system complexity (for example through metrics such as pattern redundancy). De Pauw et al. report that future work is to include improving the flexibility of the pattern matching, visual grammars, and reporting of qualitative results.

3.7.3 Evaluation

De Pauw et al. [De Pauw 1998] report that the system proved helpful for discovering unexpected behaviour, comprehension of unfamiliar code, and performance improvement in both medium-sized systems (such as Ovation itself) and large systems (such as Taligent).

3.7.4 Comparison

Walker et al. [Walker 1998] believe that the analyst requires a detailed knowledge of the system under investigation in order to compose appropriate queries for Ovation.

Koskimies and Mössenböck [Koskimies 1996] believe that the techniques employed in Scene and Ovation are complementary. They observe [Koskimies 1996] that Ovation compresses the extracted execution trace into statistical information, while Scene retains the trace. The variations in the two approaches are due to their different intended applications. Scene aims to visualise method calls and returns, whereas Ovation aims to characterise and illustrate programs using dynamic statistics. The call summary in Scene is an example of such statistical information, and was inspired by [De Pauw 1994].

3.7.5 Assessment

As with the other method-level visualisation tools, it would be expected that Ovation would perform better in a specific reverse engineering task, where the area of application would be more focussed than in a general software comprehension task. However, the summary views of Ovation may be useful in this latter context.

3.8 Reflexion models

3.8.1 Overview

3.8.1.1 AVID

Availability: http://www.cs.ubc.ca/~murphy/AVID

Level of abstraction: 4

Analysis language: (1) Smalltalk
(2) Java

Implementation language: (1) Smalltalk
(2) Java

Platform: (1) Smalltalk environment
(2) Sun JVM

3.8.1.2 RMTool

Availability: http://www.cs.ubc.ca/~murphy/jRMTool/doc

Level of abstraction: 4

Analysis language: Retargetable

Implementation language: (1) C++, Tcl/Tk
(2) Java

Platform: (1) MS Windows, UNIX
(2) Sun JVM

3.8.2 Description

3.8.2.1 AVID

Walker et al. [Walker 1998] describe an approach for producing architectural–level dynamic visualisations. The approach derives its abstractions from the number of objects in the program trace, and the communications between these objects. The tool uses a sequence of cels to represent the information collected during the system’s execution. Each cel constitutes an abstraction of dynamic information about the system at that point, and about the execution until that point. The approach is intended to complement and extend existing techniques for analysing dynamic information. The benefits of the approach are that it enables the analysis of a system without changing the source code, allows the user to manipulate the abstraction, provides an offline visualisation that is independent of the execution speed of the target system, and allows the analyst to navigate both forwards and backwards through the visualisation. The tool was originally implemented in Smalltalk for the analysis of Smalltalk programs, and has since been implemented in Java for the analysis of Java programs and named AVID (Architectural Visualization of Dynamics in Java Systems).
The tool has two main views. One view displays a series of cels showing the events that occurred during the program execution. The other view is a summary view showing cels representing an aggregate of the whole execution. The execution can be viewed in an animated form in the first view, and the user can step both forwards and backwards through the execution. Each cel consists of: a box that represents a set of objects in the high-level model defined by the analyst; a directed hyperarc passing between and through a number of boxes; a set of directed arcs between pairs of boxes, representing method calls; a histogram representing the age and garbage collection status of the objects associated with the box; annotations and bars within boxes; and annotations on each directed arc. The hyperarc represents the call stack at the end of the interval displayed. The summary view is equivalent to the final cel of the animated view. Additionally, it displays two histograms for each box: one showing the pattern of object allocation for the entire execution, and the other the age of garbage-collected objects. An example of a summary view is shown in Figure 4.30. Although only one view can be displayed at a time, the offline nature of the tool allows multiple instances to be run simultaneously on the same execution trace. The animation controls allow the user to “play” the trace, step back and forward through it, and set the step (number of cels between steps) and interval (number of events represented per cel) size. Clicking an arc, hyperarc, or histogram in either view pops up a text box giving more information on the selection. Walker et al. [Walker 1998] note that it would be possible to link the tool with a textual code browser, and have the browser jump to the relevant position in the source code when an item in the text box popup is selected.

Constructing a visualisation is a four-stage process. Firstly, execution data is extracted from the system under analysis and stored to disk. Secondly, the analyst produces a high-level model of the system using abstract entities designed to emphasise the architectural properties that he is investigating. Thirdly, the analyst defines a mapping from the abstract entities to the extracted dynamic information. The tool then applies this mapping to the extracted information to produce the visualisation. Finally, the analyst examines the visualisation to investigate the system’s dynamic behaviour. This offline, multi-stage process increases the tool’s usability by allowing iterations over the latter stages of the process – there is no need to re-run the program to collect the dynamic information again. This process is based on the concept of reflexion models introduced by Murphy et al. [Murphy 1995].

The tool collects information for every method call, object creation, and object deletion, which consists of the class of the calling (or creating) object, and either the method being called and the class of the object containing it, or the class of the object being created or deleted. The tool was originally implemented in Smalltalk and the dynamic information is collected by instrumenting the Smalltalk VM. A map relates dynamic system entities (e.g. objects or methods) to abstract ones (e.g. a box in the visualisation). The mapping process is achieved by use of regular expressions. The map consists of a set of entries, each with three parts: the name of the level of the Smalltalk structural hierarchy being mapped (i.e. application, subapplication, category, class, or method); a regular expression defining the set of names to be mapped for that level; and the name of the abstract entity to which the entities represented by these names should be mapped.
As discussed in Section 2.3, the separation of visualisation from system execution by using an off-line approach has two benefits. Firstly, pre-processing can be performed prior to visualisation, e.g. to generate summary information for the entire execution. Secondly, it allows the trace to be replayed from an arbitrary point without having to re-run the execution. Concerning navigation, a further advantage of the off-line approach is that the user can play, step back and forward through, and access randomly any part of the execution. Although no information on execution time is built into the representations, Walker et al. [Walker 1998] note that this could be desirable. An object is identified by a description of the call stack that exists when the object is created.

An area for further research is the possibility of allowing objects’ mappings to change, to allow them to “migrate” between abstraction units. Walker et al. [Walker 1998] recognise the difficulties concerning the huge volume of data generated by tracing, and believes that the flexibility and usability of the tool is limited by the use of trace information, and that the use of sampled information could partially resolve such limitations.

3.8.2.2 RMTool

Murphy et al. [Murphy 2001] discuss a technique to extract a model of a system that is “good enough” to be used for a specified task. The reflexion model technique involves comparing a high-level model (produced by the analyst) of a system with the actual implemented model. The analyst defines a mapping (using regular expressions) between the source code constructs (e.g. file names, class names, function names, etc.) and his high-level model. The RMTool (Reflexion Model Tool) system compares the two models and produces a diagram containing the modules from the analyst’s model with three types of arc connecting them: convergences (communications that agree with the analyst’s model), divergences (communications that did not appear in the analyst’s model, but do appear in the extracted model), and absences (communications that appear in the analyst’s model but not in the extracted model). Examples of corresponding high-level and reflexion models are shown in Figure 4.26 and Figure 4.27 respectively. Not all source code constructs need be mapped to a high-level equivalent – partial and approximate models are allowed. The process is designed to be iterative – the mapping can be refined as the task proceeds. Murphy et al. [Murphy 2001] give the key characteristics of the technique as being that it is “lightweight”, requiring low effort, and a timeframe of hours, not days, “approximate”, using a variety of source models and refining the mapping as the analysis proceeds, and “scalable”, capable of analysing various languages, and systems from several to over 1000 KLOC. The procedure is as follows: the analyst specifies his model; he then uses a third-party tool to extract structural information from the system (via static or dynamic analysis); he then defines the mapping between this source model and his high-level model; the analyst uses a tool to compute the reflexion model; and finally he investigates the reflexion model via a GUI.

A formal Z specification of the technique for producing the reflexion models is given in [Murphy 2001]. Optimisations were applied to reduce the computation time to acceptable levels (55 seconds for the 1000 KLOC MS Excel application). Murphy et al. [Murphy 2001] discuss the similarities and differences between their tool and
consistency checkers, reverse engineering tools, knowledge-based approaches, and model comparison techniques. Future work is to include use of the tool to produce documentation on demand for a specific task.

3.8.3 Evaluation

3.8.3.1 AVID

A qualitative evaluation was obtained through two case studies involving performance-tuning tasks on Smalltalk programs, each involving an expert and a non-expert Smalltalk developer. The expert participant found the summary view and animated hyperarc useful, but that the tool was lacking integration with a traditional code browser and the ability to view a detailed stack dump as in a Smalltalk debugger. The tool was designed to allow the integration of a code browser, but seeks to complement existing techniques, so does not seek to replace a debugger by incorporating one. The non-expert found the garbage collection histograms, and the correlation between abstract information and method/object names available in the pop-up useful, but desired different displays of information, feeling that one screen was “too cluttered”.

3.8.3.2 RMTool

Murphy et al. [Murphy 2001] discuss the tool in the context of NetBSD (written in C), and a number of case studies are discussed, including Microsoft Excel (C), the SPIN OS (Modula-3 and C), and a restructuring tool (C++); the tool appeared to help with all of them.

The MS Excel case study is described in more detail in [Murphy 1997]. The Excel application consists of 1.2 MLOC of C. The goal of the reengineering task was to identify and extract components from the application source code. To achieve this, an understanding of the structure of the application was required. Specifically, the analyst needed to gain an understanding of how the source code was divided into static modules, and how the modules communicated at runtime. The analyst reported that the reflexion model technique had assisted him in refining an architectural view of the application, and in investigating the correspondence between that view and the source code. Additionally, the reflexion model helped the analyst with his overall understanding of the application, and highlighted aspects that were not apparent from the initial high-level model or the source code. The analyst also reported that it was straightforward to focus the investigation on the relevant parts of the system and exclude extraneous detail. Murphy and Notkin [Murphy 1997] assert that this case study proves that the reflexion model technique has useful practical applications for the following reasons. Firstly, the analyst elected to use the reflexion model technique even with the constraints of an industrial setting. Secondly, the analyst continued to use the technique for future revisions of the application outwith the case study period. Thirdly, the analyst believed that the reengineering task could have been completed sooner had the reflexion model technique been employed earlier. Murphy and Notkin [Murphy 1997] attribute much of the success of the technique to its support for approximation in the form of unrefined areas of the model. They believe that the
results of this case study can be generalised to similar reengineering efforts, as the application was written in a commonly-used language (C), the source code had evolved over time with multiple developers, and the task of identifying and extracting components from an existing system is a common one.

3.8.4 Comparison

Richner and Ducasse [Richner 1999] note the similarity of their process with that of Murphy and Notkin [Murphy 1997], in that it allows the analyst to navigate their investigation through an iterative process. Another similarity is that [Richner 1999] also expects the engineer to produce a high-level model of the system under analysis.

Richner and Ducasse [Richner 1999] believe that the Gaudi technique complements that of Walker et al. [Walker 1998] in recognising that object-level tracing information is too low-level to assist in architectural understanding of a system. While [Walker 1998] appears to be targeted to performance evaluation, Gaudi aims to allow the analyst to specify the view that most suits his analysis.

Systä et al. [Systä 2001] observe that in [Walker 1998] the mapping between low-level system artefacts and high-level components of the analyst’s model is constructed manually using a declarative mapping language. Shimba presents static and dynamic information in separate views, and Rigi is used to build high-level static components. The analyst can then construct high-level sequence diagrams by mapping low-level artefacts to high-level components.

Murphy et al. [Murphy 2001] present the idea of combining models from different extractors as a simple case of set union, which is in contrast to the production of fused views in Dali described in [Kazman 1997]. A possible disadvantage of the reflexion model technique is that the analyst needs to start with a model – the system gives no help if the model is very inaccurate. It must be considered whether or not it would always be acceptably straightforward to produce a sufficiently accurate model. The technique appears to require either an understanding of the system under investigation, or an experienced analyst. The effort involved in producing the mapping for a large system would appear to be considerable, even if it were produced iteratively (e.g. 1,425 map entries for Excel). The system appears to be reliant on conventions (e.g. directory or class structure) in the source code for producing its models; although Murphy et al. [Murphy 2001] note that this was not a problem in their case studies, if the source code is disorganised, the model produced may be of little value.

3.8.5. Assessment

The high-level architectural views produced by these tools would be expected to be useful in general software comprehension tasks, provided appropriate high-level models of the target system could be constructed. The reflexion model approach may be less successful with specific reverse engineering tasks, depending on the level of abstraction required.
3.9 *Gaudi*

### 3.9.1 Overview

**References:** [Richner 1999]

**Availability:** Not available

**Level of abstraction:** 3 – 4

**Analysis language:** Retargetable

**Implementation language:** SICTUS Prolog, Smalltalk

**Platform:** Smalltalk environment

### 3.9.2 Description

Richner and Ducasse [Richner 1999] describe a technique for extracting application visualisations using a combination of static and dynamic information. A set of Prolog facts defines the basic static (e.g. superclass-subclass) and dynamic (e.g. message send) relations between elements. Derived relations can be produced from these, such as *overrides* (static) and *sendsCreate* (dynamic). Views are defined by a describing a set of components and the connectors between them. Prolog rules are used to define a clustering of components (C), and a relation (R). The diagrams contain ovals representing components, and directed arcs representing communications between those components. Methods can also be grouped by class.

The static information is extracted by parsing the code using the *MOOSE* tool [Ducasse 2000] and representing it in the *FAMIX* model [Tichelaar 1998]. The dynamic information is collected by instrumenting the application with Method Wrappers [Brant 1998], and stored as Prolog facts. Prolog queries are used to build the abstractions. The *Gaudi* tool was used to create the views, which were then displayed using the *dot* tool [Koutsofios 1996]. Richner and Ducasse [Richner 1999] note that the approach could be adapted easily to Java to C++, but that it does not presently support concurrency.

Richner and Ducasse [Richner 1999] give the weaknesses of the approach as follows. Obtaining dynamic information requires an executable, instrumentable system – *Gaudi* is therefore not suitable for sections of partially constructed systems, or other unexecutable code. They also note the problem of scalability, and gives possible solutions as instrumenting only some methods/classes, feedback from query results to instrumentation so that only relevant methods are instrumented, appropriate scenario choice, and pre-analysis trace filtering.

Richner and Ducasse [Richner 1999] give the strengths of *Gaudi* as flexibility in the kinds of views that can be recovered by allowing the analyst to define relations and clusterings and in the questions that can be answered through its use of both static and dynamic information.
They identify future work as including determining which views are most useful in reverse engineering, and on guidelines for the use of such tools in reverse engineering.

3.9.3 Evaluation

A case study of reverse engineering of Smalltalk HotDraw [Johnson 1992, Beck 1994] demonstrates the technique. The case study proceeded as follows. A high level view was created that shows all the relations between HotDraw classes, grouped by Smalltalk category. Based on this information, a new clustering was then defined to give a different view. A view was then created showing creation invocations, and one to show non-creation invocations.

The clustering in Gaudi provided a number of views at different levels of granularity, while the combination of static and dynamic information was reported to assist in focussing the effort. The views produced helped the analyst to formulate questions about the interactions in the system, and provided a comparison with his own mental model of the system.

3.9.4 Comparison

Systä et al. [Systä 2001] observe that the query-based approach of Gaudi allows the user to tailor the views produced, which may contain either static or dynamic information, or a combination of both, and exist at various levels of abstraction. The query-based approach also allows the analyst to control the volume of information generated. However, unlike Shimba, Gaudi does not support the direct exchange of information among views.

3.9.5 Assessment

In common with other architectural-level tools, Gaudi would be expected to perform best in general software comprehension tasks. The varying levels of abstraction that can be produced using its query-based approach may also allow it to perform well in specific reverse engineering tasks.

3.10 Shimba

3.10.1 Overview


Availability: Not available

Level of abstraction: 2 – 4
Analysis language: Java

Implementation language: Java

Platform: Sun JVM

3.10.2 Description

Systä et al. [Systä 2001] describe the Shimba tool, which produces visualisations of Java programs using both static and dynamic information. Shimba extracts static and dynamic information from the Java bytecode of the system. It displays static information using directed graphs (Rigi dependency graphs), and dynamic information using a variation of UML sequence diagrams (SCED (Scenario Editor) sequence diagrams) from which statecharts can be generated automatically. The principal contribution of [Systä 2001] is that Shimba considers both static and dynamic information and constructs separate diagrams for each, but maintains a relationship between the diagrams. Most other tools consider either static or dynamic behaviour, or combine both into a single diagram.

The dynamic information is extracted by running the target system under a customised Java SDK debugger [Sun 2000], which automatically sets breakpoints in the code. Shimba integrates the Rigi [Müller 1988, Müller 2001] (static) and SCED [Koskimies 1998] (dynamic) tools to carry out both general program understanding and goal-driven reverse engineering. Shimba (and, in a similar manner, Dali) demonstrates the possibility of constructing software comprehension tools using pre-existing tools, rather than starting from scratch. SCED sequence diagrams can be used to slice the static graphs produced by Rigi, to enable visualisation of the part of the system that is responsible for a particular observed behaviour. Rigi graphs can be used to guide the generation of SCED sequence diagrams to observe the behaviour of a specific part of the system, and can also be used to raise the level of abstraction of the SCED diagrams. Dynamic control flow information can also be added to sequence diagrams, while the static graphs can be annotated with software metrics [Chidamber 1994]. The event trace explosion problem is handled by applying behavioural pattern matching algorithms [Boyer 1977] to the trace to extract out repeated patterns. These are then represented in the SCED sequence diagram using subscenario and repetition constructs. The trace can be split (both automatically and by the user) into a number of smaller traces to limit the size of the sequence diagrams produced.

Systä et al. [Systä 2001] note that the techniques in Shimba are also applicable to forward engineering, to check the implemented structure against design guidelines, and the implemented behaviour against use cases. Future work will integrate the techniques of Shimba into the Nokia TED UML modelling tool [Wikman 1998]. This will allow the usefulness of the techniques in Shimba to be studied with real users, and will allow tighter integration than is possible with current reverse engineering environments. Systä et al. [Systä 2001] comment that a reverse engineering environment using various UML diagrams would be useful.

Further details on the use of Shimba in analysing metrics is given in [Systä 2000a]. Further information on the reverse engineering of Java software using Shimba is given.
in [Systä 2000b, Systä 2000c], using Rigi and SCED in [Systä 1999a], and using SCED in [Systä 1999b] and [Systä 2000d].

3.10.3 Evaluation

A case study of the FUJABA system [Rockel 2000] illustrates the use of Shimba. The combination of static and dynamic information was found to be particularly useful. Although the string matching algorithms employed were able to detect numerous, nested patterns in the trace, one of the most problematic aspects involved structuring the SCED sequence diagrams using behavioural patterns. One problem related to the naming of subscenario boxes, which is automatic and therefore not descriptive of the subscenario. Another problem relating to subscenarios was that a pattern is defined based on its length and contains an arbitrary sequence of SCED sequence diagram elements, which may not necessarily form a logical unit within the context of the system under analysis.

Using static information to guide the generation of dynamic information was found to be particularly useful for goal-driven reverse engineering tasks. This helps to prune ‘uninteresting’ information from the visualisation. The statechart synthesis functionality was useful for analysing the dynamic behaviour and control flow of selected parts of the system. The model slicing technique was used to determine the cause of certain behaviour, the system structure that relates to this behaviour, and how elements of a SCED sequence diagram relate to the rest of the system. Raising the level of abstraction of the SCED sequence diagrams using static Rigi abstractions was also employed to understand communication between high-level components, and to validate such static abstractions.

Further information on this case study is given in [Systä 2000c].

3.10.4 Comparison

Unlike other tools that produce diagrams containing only static or dynamic information, or combine both into one diagram, Shimba produces separate diagrams for static and dynamic information and provides linkages between them. The pattern-matching functionality is comparable to that employed in Ovation, allowing repeated behaviour to be factored out as a subscenario in the visualisation. Shimba’s automatic statechart generation function is unique – none of the other tools considered produce state-level representations of dynamic behaviour.

3.10.5 Assessment

The sequence diagram, statechart, and dependency graph representations used in Shimba should enable it to perform well in specific reverse engineering tasks. The ability slice the static dependency graphs using dynamic sequence diagrams, and to raise the level of abstraction of a scenario diagram using high-level static abstractions should make Shimba useful for general software comprehension tasks also.
3.11 Jinsight

3.11.1 Overview


Availability: http://www.alphaworks.ibm.com/tech/jinsight

Level of abstraction: 2 – 3

Analysis language: Java

Implementation language: Java

Platform: IBM AIX, ISM OS/390, MS Windows

3.11.2 Description

De Pauw et al. [De Pauw 2002] describe the Jinsight tool and its application to the visual exploration of runtime information. Jinsight illustrates object population, thread activity, and method calls Java software. Jinsight includes a profiling agent that is used to produce an execution trace from which visualisations are generated. Tracing can be enabled and disabled during execution. Uninteresting classes and packages can be excluded from the visualisation. Visualisations are presented in the form of interdependent views, each of which illustrates a different facet of the software’s runtime behaviour.

One such view is the histogram view, which illustrates resource usage (CPU time and memory space) for classes, objects, and methods. This view allows the analyst to identify ‘hot spots’ of activity in the execution that could indicate a bottleneck. Each row in the histogram corresponds to a class in the system. Symbols are coloured to represent activity on that class, such as the time spent executing methods of the class, the number of calls made to methods in the class, the amount of memory consumed by instances of the class, or the number of threads in which instances of the class participate. Hollow rectangles represent garbage-collected objects, which helps in identifying memory leaks. The lines in the histogram view represent inter-object communication, and can be set to show either method calls, object creation, or references between objects. However, the combinatorial nature of inter-object communications means that this aspect of the histogram view is not scalable beyond very simple programs.

One way in which Jinsight simplifies the huge amount of data produced from a dynamic trace is through pattern extraction. A patter extractor analyses the event trace information and identifies patterns of repeated behaviour. These patterns can be used to present an aggregated view of the execution. The reference pattern view illustrates

---

1 Hot spots in this context are distinct from hot spots in the context of framework reuse, where they are points where a framework is designed to be extended.
patterns of object references in the execution. Colours denote classes. Double rectangles represent a group of objects of a certain type. Labels denote the number of instances of a class, and the class name. The reference pattern view can be used to help in identifying memory leaks in the form of objects that are no longer required but cannot be garbage-collected due to outstanding references from other objects.

*Jinsight*’s execution view illustrates the sequence of method calls that make up the event trace of the system’s dynamic behaviour. An example execution view is shown in Figure 4.11. Time (indicated on the right hand side) proceeds from top to bottom. Stripes are coloured by class. Each stripe represents the execution of a method, with deeper calls at the right hand side. A lane constitutes all of the method stripes for a thread of the execution. Lanes are added from left to right (though only one lane appears in Figure 4.11). Zooming in further to the execution view reveals individual method calls, annotated with their names. Pattern recognition can also be applied to the execution view. The execution pattern view illustrates patterns of method calls in the execution.

The call tree view gives quantitative data on the sequence of method calls, including the number of calls, and their contribution to the total execution time.

*Jinsight* allows the analyst to group related behaviour into execution slices, which can be used as a basis for comparison between executions, or to filter out information not pertinent to the visualisation objectives. Execution slices can be defined by selecting elements in a view, or by querying the trace data directly.

De Pauw and Sevitsky [De Pauw 1999, De Pauw 2000] describe the use of *Jinsight* in examining memory leaks, while Sevitsky et al. [Sevitsky 2001] discuss the use of *Jinsight* for performance analysis. A brief summary of *Jinsight*’s functionality is given in [De Pauw 2001].

Future work includes enabling the visualisation of systems running on multiple JVMs simultaneously and across networks, and of heterogeneous systems containing middleware such as databases in addition to Java components.

### 3.11.3 Evaluation

De Pauw et al. [De Pauw 2002] report that *Jinsight* has been used successfully to diagnose a number of problems in industrial applications. They note that the system did not perform well when analysing high-volume web-based applications as the tracing overhead caused undesirable behaviour in the application, requiring more selective trace information collection. They found that their aggregate statistics did not provide sufficient information to support some analyses, and that broad filtering at the class or method level did not scale well. To rectify this, *Jinsight* allows task-oriented tracing, where relevant details can be extracted while retaining other important contextual information.
3.11.4 Comparison

De Pauw et al. [De Pauw 2002] note that it is important to select appropriate diagram abstractions that are sufficiently scalable to large amounts of execution information. They comment that Sefika et al. [Sefika 1996] use large architectural units, while Walker et al. [Walker 1998] include additional structural units to organise the data.

*Jinisight* shares some ideas with *Ovation* [De Pauw 1998], notably the concept of execution patterns.

3.11.5 Assessment

The call tree view and execution view would be expected to help with specific reverse engineering tasks. The reference patterns may be useful for general software comprehension tasks. *Jinisight* would appear to be particularly useful for examining performance issues, for which the histogram view would be useful.

3.12 Collaboration Browser

3.12.1 Overview

References: [Richner 2002a, Richner 2002b]

Availability: Not available

Level of abstraction: 2 – 4

Analysis language: Smalltalk

Implementation language: Smalltalk

Platform: Smalltalk environment

3.12.2 Description

Richner and Ducasse [Richner 2002a] describe a process for recovering collaborations from software systems using dynamic information. A tool called the *Collaboration Browser* illustrates the technique. A *collaboration* represents a part of the software system that performs some function and details how the classes that make up the collaboration interact by playing certain *roles*.

The first stage in extracting collaborations from source code is to dynamically analyse the code to extract interactions. Static analysis is inadequate for this purpose as it cannot provide the object-oriented control flow information required. It is then necessary to identify the important collaborations that help to answer the analyst’s questions. *Collaboration Browser* records an event trace containing information for each method call, consisting of sender class and identity, receiver class and identity,
and the name of the called method. Pattern matching is used to abstract similar sequences of execution from the trace. Querying allows the analyst to identify the interesting collaborations.

A collaboration instance is the sequence of method calls between a method call and its corresponding return. A collaboration pattern is a generalised class of collaboration instances, and represents the collaboration design concept. The set of methods called on a class during a collaboration pattern correspond to the role design concept.

The pattern matching settings used to identify collaboration patterns from instances can be adjusted in three ways. Firstly, any of the five items of information that represent an event in the trace (caller class and identity, callee class and identity, and method) can be included or excluded from the match. Secondly, events can be ignored when an object sends itself a message, or if the depth of invocation exceeds some limit in the pattern or overall execution. Thirdly, the analyst can choose to treat events as a tree-structure sequence, or simply as a set of events with no implied ordering.

Collaboration Browser uses a textual GUI to allow the analyst to query the entire execution or a single collaboration. The analysis can be focussed by excluding selected senders, receivers, or methods. A collaboration can also be illustrated as a sequence diagram.

Two limitations of the Collaboration Browser were identified as follows. Firstly, the pattern matching was simplified by only considering all of the events between a method call and return; it could be useful to consider a subset. Secondly, the role of a class is identified as the set of all methods called on that class during the execution; considering individual class instances separately could produce a more refined view of roles.

Collaboration Browser is implemented in Smalltalk and visualises Smalltalk programs. The program to be analysed is instrumented using Method Wrappers [Brant 1998], which allows selective instrumentation. The Interaction Diagram tool [Brant 1998] is used as the basis for the sequence diagram representations.

Richner and Ducasse note that the recovery of collaborations is most effective when combined with high-level views showing the interaction of components in a system [Richner 1999, Richner 2002b].

3.12.3 Evaluation

Collaboration Browser is evaluated in a HotDraw case study where the goal is to investigate the implementation of tools. The scenario executed produced 53,735 method calls, from which 183 collaboration patterns were extracted using the pattern matching functionality. The results were then queried to discover the collaboration patterns containing an interaction between the Tool class and another class in the trace; this produced twelve unique collaboration patterns. The results were then focussed further to examine four collaboration patterns resulting from a call to Tool.handleEvent. Further queries on these collaboration patterns revealed the role
played by each of the participant classes. The role of Tool in other collaborations was also investigated. Further case study evaluation of Collaboration Browser is given in [Richner 2002b].

It is reported that the case studies showed that the queries helped in locating interesting collaborations and in understanding the roles of classes in collaborations. They also demonstrate that the process cannot be fully automated – a human analyst is required. It was a challenge to identify suitable pattern matching criteria to obtain a balance between too much and too little information. The iterative process employed in the case study was as follows: collaboration patterns were created; queries were formulated regarding class interfaces; collaboration patterns involving certain classes were identified; the collaboration pattern participants were investigated; and the collaboration was investigated further using the interaction diagram representation.

3.12.4 Comparison

Richner and Ducasse [Richner 2002a] consider their approach to be complementary to other reverse engineering techniques that are more focussed towards visualisation, such as those of De Pauw et al. (Ovation and Jinsight) [De Pauw 1998] and ISVis [Jerding 1997]. The approach of Richner and Ducasse is focussed more on querying the trace data to extract collaborations than on producing a visualisation. They feel that, whereas the techniques of De Pauw et al. and Jerding and Rugaber consider the trace as a whole, their approach complements these techniques by concentrating on smaller portions of the interaction. They also note that no single tool can provide all of the functionality necessary for design recovery.

The only other approach that attempts to reverse engineer collaborations is one based on static analysis only [De Hondt 1998]. This approach relies on the analyst selecting participants and roles for the collaboration and proposing appropriate links between them.

3.12.5 Assessment

The collaboration approach used in Collaboration Browser suggests that it would be useful in general comprehension of specific parts of a software system. The ability to view collaborations as sequence diagrams would be expected to be helpful in specific reverse engineering tasks.

3.13 Together debugger

3.13.1 Overview

References: [TogetherSoft 2001a, TogetherSoft 2001b]
Availability: http://www.togethersoft.com/products/controlcenter
Level of abstraction: 1
Implementation language: Java
Platform: Compaq True64 UNIX, HP-UX, Linux, MS Windows, Sun Solaris

3.13.2 Description

The Together debugger is part of the Together ControlCenter development environment [TogetherSoft 2001a, TogetherSoft 2001b]. It provides all of the standard debugger features, including breakpoints, expression evaluation and monitoring, variable modification, and program flow control. Breakpoints can be set at classes, methods, lines, or exceptions. Whenever a breakpoint is encountered during the execution of the program, the debugger outputs a message and/or suspends the execution. The values of variables and expressions can be monitored during execution, and variable values can be modified. Program execution can be suspended and resumed by the user. Execution can proceed as normal, or in steps, where the debugger executes one line of code then suspends. The user can instruct the debugger to step to the next line, or into, out of, or over a method. Integration with the source code allows the user to set breakpoints and watches by selecting a position in the code, and also to instruct the debugger to run the program up to the current cursor position. The debugger interface is shown in Figure 4.33.

3.13.3 Evaluation

There do not appear to have been any evaluations published regarding the performance or functionality of the Together debugger.

3.13.4 Comparison

The graphical interface of the Together debugger makes it easier for non-experts to use. Debuggers traditionally have a command line interface, for example jdb [Sun 2002]. The integration with the source code also makes it more convenient to set and manage breakpoints and watches.

3.13.5 Assessment

The low level information provided by the debugger is likely to be useful for some specific reverse engineering tasks, which are often amenable to analysis at a low level of abstraction. The debugger is less likely to be useful for general software comprehension tasks, where information at a higher level of abstraction is typically required.
3.14 Together diagrams

3.14.1 Overview

References: [TogetherSoft 2001a, TogetherSoft 2001b]

Availability: http://www.togethersoft.com/products/controlcenter

Level of abstraction: 2 – 3


Implementation language: Java

Platform: Compaq True64 UNIX, HP-UX, Linux, MS Windows, Sun Solaris

3.14.2 Description

Together ControlCenter can produce UML class and interaction diagrams from program source code. Unlike the other tools considered in this section, Together produces dynamic diagrams by parsing the program code, rather than by analysing an event trace. As discussed in Section 2.1, this limits the accuracy of the interaction diagrams generated, while maximising their generality by considering the entire system. This is an example of dynamic information that is not dynamically extracted (see Section 2.1.1). When generating interaction diagrams, Together addresses the potential information overload problem inherent in dynamic analysis by allowing the user to select the classes to be included in the diagram, limit the depth of method calls to be included, and hide method internals. Interaction diagrams are generated for a method specified by the user. Together supports ‘simultaneous round trip engineering’, meaning that changes to the program code are reflected in the derived diagrams and vice versa. Figure 4.7 shows a class diagram generated by Together; Figure 2.1 and Figure 2.2 show corresponding sequence and collaboration diagrams generated by Together.

3.14.3 Evaluation

Kollmann et al. [Kollmann 2002a] present a comparison of four static reverse engineering tools. Together is compared with the commercial Rational Rose tool [Rational 2003], and the IDEA [Kollmann 2001, Kollmann 2002b] and Fujaba [Fujaba 2002] research tools. As this was a static case study, only the class diagram generation facility of Together was examined. The tools were assessed by evaluating the class diagrams that they produced. While basic diagram generation results were broadly similar across the tool set, Rational Rose detected some associations that Together did not. The research tools were able to handle more advanced diagram concepts than the industrial tools, such as multiplicities, inverse associations, and container resolution.
3.14.4 Comparison

Together is unique among the tools in this section as it produces dynamic diagrams by parsing the program code. All other tools considered extract dynamic information from an event trace produced during program execution. As discussed above, this has the effect of reducing the detail of the diagrams while increasing their generality.

3.14.5 Assessment

It would be expected that the combination of the class and interaction diagrams for the entire system produced by Together would be useful in general software comprehension tasks. The lack of dynamically extracted information and resultant lack of detail may be a problem in specific reverse engineering tasks.

3.15 Tool summary

This section has reviewed a selection of dynamic visualisation tools, which illustrate the concepts described in Section 2. Each tool was discussed in the context of the three characteristic criteria introduced in Section 3.1.1. Table 3.1 summarises the overviews of the tools reviewed. Early object-oriented dynamic visualisation tools were concerned primarily with illustrating method-level interactions; such tools included Program Explorer and Scene. Later tools began to consider the problem of architectural extraction, and architectural-level visualisations were produced by tools such as [Sefika 1996a], ISVis, Dali, AVID, and RMTool. The latest tools have attempted to bridge the gap between microscopic and macroscopic visualisations and provide both low-level and architectural visualisations, namely Gaudi, Shimba, and Collaboration Browser. Tools have also been developed to address specific tasks, such as Jinsight for performance analysis, and Together to support software development. Figure 3.2 annotates the scale from Section 3.1.2 to illustrate the relative levels of abstraction of these tools. There is an emerging trend of retargetable dynamic visualisation tools, which can be used to visualise programs in a variety of languages, rather than being designed for use with one specific language. Such tools include Dali, RMTool, Gaudi, and Together. A retargetable design makes the tool more flexible and should encourage usage and interoperability. The following section will assess those tools that are available in the context of a case study involving both general software comprehension and specific reverse engineering activities.
Figure 3.2. The positions of tools on the abstraction scale of Figure 3.1.
### Table 3.1 Tool overview summary

<table>
<thead>
<tr>
<th>Tool</th>
<th>Abstraction level</th>
<th>Analysis Language</th>
<th>Implementation Language</th>
<th>Platform</th>
<th>Availability</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Explorer</td>
<td>2</td>
<td>C++</td>
<td>C++</td>
<td>IBM AIX</td>
<td>Not available</td>
<td>[Lange 1995a, Lange 1995b, Lange 1997]</td>
</tr>
<tr>
<td>Architecture-oriented visualization</td>
<td>4</td>
<td>C++</td>
<td>Berkeley YACC, SWI Prolog, Tcl</td>
<td>Unix</td>
<td>Not available</td>
<td>[Sefika 1996a]</td>
</tr>
<tr>
<td>ISVis</td>
<td>4</td>
<td>C++, C++</td>
<td>C++</td>
<td>Sun Solaris</td>
<td><a href="http://www.cc.gatech.edu/morale/tools/isvis/isvis.html">http://www.cc.gatech.edu/morale/tools/isvis/isvis.html</a></td>
<td>[Jerdling 1997]</td>
</tr>
<tr>
<td>AVID</td>
<td>4</td>
<td>Smalltalk</td>
<td>Smalltalk</td>
<td>Sun JVM</td>
<td>Not available</td>
<td>[Murphy 1995, Murphy 1997, Walker 1998, Murphy 2001]</td>
</tr>
<tr>
<td>RMTool</td>
<td>4</td>
<td>Retargetable</td>
<td>C++, Tcl/Tk</td>
<td>MS Windows, UNIX</td>
<td>Not available</td>
<td>[Richner 1999]</td>
</tr>
<tr>
<td>Collaboration Browser</td>
<td>2–4</td>
<td>Smalltalk</td>
<td>Smalltalk</td>
<td>Not available</td>
<td>[TogetherSoft 2001a, TogetherSoft 2001b]</td>
<td></td>
</tr>
<tr>
<td>Together diagrams</td>
<td>2–3</td>
<td>Java</td>
<td>Java</td>
<td>IBM AIX</td>
<td>Not available</td>
<td>[TogetherSoft 2001a, TogetherSoft 2001b]</td>
</tr>
</tbody>
</table>
4 Case study

4.1 General questions

This case study will evaluate the available tools by assessing their performance in a number of dynamic visualisation tasks. The tasks take the form of questions that an analyst would find it useful to be able to ask about a software system. Large-scale questions consider the entire system, and are typical of those that would be asked in a general software comprehension effort. Small-scale questions address only a part of the system, and are typical of those asked while carrying out a specific reverse engineering task. These general questions can be reused for the evaluation of any type of software comprehension tool in the context of any specific system. The large-scale questions are immediately reusable, while the small-scale questions can be instantiated within the context of the system being used for the evaluation. The JHotDraw semantic drawing editor framework [Gamma 1998] was chosen for this case study as it a reasonably complex, real-life application framework typical of the type of system that would be subject to software comprehension and reverse engineering efforts. HotDraw is also widely used as a case study in the literature.

The tools evaluated were Together diagrams, Jinsight, jRMTool, AVID, and Together debugger.2 All tasks were carried out on a minimally loaded AMD Athlon XP 2100+ machine with 512MB RAM running Windows 2000 Professional.

4.1.1 Large-scale questions

The following questions are intended to be typical of those asked during the course of a software comprehension effort. Questions L1-L6 are inspired by the six ‘overall understanding’ questions of Systä et al. [Systä 2001, p.378]. Questions L7 and L8 address issues that are particularly relevant to framework reuse, while L9 is an important software comprehension issue.

L1 What is the static structure of the software system?
L2 What interactions occur between objects at runtime?
L3 What is the high-level structure/architecture of the software system?
L4 How do the high-level components of the software system interact?
L5 What patterns of repeated behaviour occur at runtime?
L6 What is the load on each component of the software system at runtime?
L7 What design patterns are present in the software system's implementation?
L8 Where in the software system are the hotspots where additional functionality can be added?
L9 What impact will a change made to the software system have on the rest of the software system?

2 Although Dali is retargetable, a tool to provide dynamic information from Java programs in the format required by Dali was not available.
4.1.2 Small-scale questions

The following questions are intended to be typical of those asked during the course of a specific reverse engineering effort. Questions S1, S2, and S6 are inspired by the ‘goal-driven reverse engineering’ and ‘object/method behaviour’ questions of Systä et al. [Systä 2001, p.378]. Questions S3, S4, and S5 address issues typically encountered in framework comprehension [Kirk 2001] and are typical maintenance activities.

S1 What are the collaborations between the objects involved in an interaction?
S2 What is the control structure in an interaction?
S3 How can a problem solution be mapped onto the functionality provided by the software system?
S4 Where is the functionality required to implement a solution located in the software system?
S5 What alternative functionalities are available in the software system to implement a solution?
S6 How does the state of an object change during an interaction?

4.2 Small-scale questions specified for JHotDraw

The system used for this case study was an orrery simulation consisting of 133 classes constructed as a sample solution to a final year undergraduate software architecture assignment at the University of Strathclyde, Glasgow. A JHotDraw drawing editor consists of a drawing containing figures and connections between them, and a set of tools for creating and manipulating the drawing elements. The orrery application is shown in Fig. 4.x. The source code for the application was available in the Orbit and CH.ifa.draw.* packages. Javadoc documentation was available for the JHotDraw classes, but not for the orrery extension. The coursework assignment worksheets provided some background to the application functionality. The following questions instantiate the small-scale questions above for the JHotDraw domain.
J1 A common problem in JHotDraw applications is the display not being updated as desired when a change is made to the model. For example, attempting to move a planet (represented by an object of type `Figure`) in an orrery application may not be reflected in the display. To understand this problem, it is necessary to investigate the redraw mechanism of JHotDraw. The redraw mechanism is an interaction consisting of a sequence of object collaborations. (Answer: The correct sequence of method calls is `Figure.willChange()`, `Figure.invalidate()`, `Figure.changed()`, then `Figure.invalidate()`.)

J2 When a `Figure` object is moved or has its dimensions changed, there may be erratic changes both to this `Figure` and to other `Figure` objects to which it is connected (by `ConnectionFigure` objects). For example, an orrery application may represent three planets (as `Figure` objects) A, B, and C, and gravity between them (as `ConnectionFigure` objects), such that A is connected to B, B to C, and C to A. If the `Figure` objects are connected such that moving one planet also moves those planets connected to it, then moving A would cause C to move, which would in turn cause B to move, which would then cause A to move, resulting in an infinite loop of `Figure` movements. To understand this problem, it is necessary to investigate the way in which JHotDraw deals with cyclic constraints such as this. Interactions in JHotDraw can use an implicit or explicit control structure; the control structure used is important in solving problems such as this.
J3 JHotDraw applications often require collision detection, so that action can be taken when two figures 'collide' (i.e. overlap on the diagram). For example, in an orrery application, it may be desirable to detect when a Figure representing an asteroid crosses the connection (i.e. overlaps the ConnectionFigure) between two Figures representing a planet and a satellite orbiting that planet respectively. To understand this problem, it is necessary to investigate the mechanism by which JHotDraw determines the locations of Figures in a drawing. Collision detection is not provided natively in JHotDraw; therefore, the solution to the collision detection problem must be mapped onto the functionality available in JHotDraw.

(Answer: JHotDraw uses the concept of a ‘display box’ to define the location of a Figure. This can cause problems by constraining all Figures to be rectangles.)

J4 Question J3 describes how collision detection can be implemented in JHotDraw by testing when two Figures' display boxes overlap. For example, if the display box of a Figure representing a planet in an orrery application overlaps with that of a Figure representing an asteroid, then a collision would have occurred. In order to implement this solution, it is necessary to investigate how a Figure’s display box can be obtained, and how display boxes can be tested to determine whether they overlap. In order to implement the solution to the collision detection issue described in Question J3, it is necessary to identify the location of the required functionality in JHotDraw.

(Answer: Figure.displayBox() returns a Figure's display box as an object of type java.awt.Rectangle. Rectangle.intersects(Rectangle) can then be used to test if two rectangles intersect.)

J5 When Figures in a diagram are moved or resized, they may also be resized or moved unexpectedly. For example, when moving a planet (represent by a Figure) in an orrery application the planet may appear larger or smaller than expected, or when resizing a planet its position may change. To understand this problem, it is necessary to investigate the way in which Figures are moved and resized in JHotDraw. JHotDraw provides a number of ways of altering the position and/or dimensions of a Figure, and it is necessary to select the appropriate functionality.

(Answer: Figure.displayBox(java.awt.Point, java.awt.Point) and Figure.displayBox(Rectangle) allow both the position and dimensions of a Figure to be changed in one operation. Figure.moveBy(int, int) can be used to move a Figure without changing its dimensions.)

J6 When debugging a JHotDraw application, it may be important to examine the internal state of objects in the diagram. For example, in an orrery application, a Figure object representing a planet would contain a reference to the mass of the planet it represents. In order to extract such information, it is necessary to investigate the way in which an object’s state changes during the course of an execution.
4.3 Together diagrams

4.3.1 Large-scale questions

L1 What is the static structure of the software system?
Together generates a class diagram to show the static structure of the software system. As described in Section 3.14, the diagram is derived from the source code, rather than from an execution trace. The class diagram generated by Together for the orrery application is shown in Figure 4.2.
Figure 4.2. The class diagram generated by *Together* for the orrery application.
L2 What interactions occur between objects at runtime?

Together can generate interaction diagrams for a specified method. As with the class diagram, sequence and collaboration diagram are derived from the source code. The sequence diagram generated by Together for the main method of the orrery application (Orbit.MainClass.main()) is shown in Figure 4.3. Together took 5 minutes and 50 seconds to produce this diagram. The method call depth was restricted to 10, and only classes in the Orbit and JHotDraw packages were included.
Figure 4.3. The sequence diagram generated by Together for the Orbit.MainClass.main() method.
L3 What is the high-level structure/architecture of the software system?
Together does not support architectural extraction directly. However, the class diagram can often be used to give some idea of the high level structure of the system. The structure of the JHotDraw inheritance hierarchy, consisting of interfaces, abstract classes, then concrete classes, is evident from Figure 4.2.

L4 How do the high-level components of the software system interact?
Together does not support the extraction of high-level interactions directly. However, some high-level interactions may be evident in the interaction diagrams generated. This is not particularly evident in the case of the JHotDraw orrery application.

L5 What patterns of repeated behaviour occur at runtime?
Together does not automatically detect patterns in execution behaviour, nor does it provide runtime information in its diagrams. Some patterns of repeated behaviour may be observable from the sequence diagrams generated.

L6 What is the load on each component of the software system at runtime?
Together does not provide runtime profiling functionality. However, some information on expected load can be gleaned from the available metrics, such as Number of Members (NOM), Attribute Complexity (AC), and Weighted Methods Per Class (WMPC1 and WMPC2).

L7 What design patterns are present in the software system's implementation?
Together does not provide any functionality for identifying design patterns. Evidence for their existence would need to be gathered by the analyst manually inspecting class and interactions diagrams.

L8 Where in the software system are the hotspots where additional functionality can be added?
Together does not support the automatic detection of hotspots. An analyst may be able to gather such information by examining the class and interaction diagrams.

L9 What impact will a change made to the software system have on the rest of the software system?
The ‘Search for Usages’ function in Together allows an analyst to search the system for all locations where a particular attribute, method, class, interface, or package is used. This would allow him to evaluate the impact of a potential alteration to the system.
4.3.2 Small-scale questions

J1 (S1 What are the collaborations between the objects involved in an interaction?)
A screen redraw occurs when the contents of the screen have been changed, for example, when a figure is moved. Examining the class diagram reveals that AbstractFigure is the abstract superclass of all Figure objects in JHotDraw. AbstractFigure has a method named moveBy() which appears to be a likely candidate for moving Figure objects. The sequence diagram generated by Together for this method is shown in Figure 4.4. This diagram shows that the correct sequence of method calls for a screen redraw is AbstractFigure.willChange(), AbstractFigure.invalidate(), AbstractFigure.changed(), then AbstractFigure.invalidate().

![Figure 4.4. The sequence diagram generated by Together for the AbstractFigure.moveBy() method.](image)

J2 (S2 What is the control structure in an interaction?)
Figure 4.4 illustrates the implicit invocation control structure of the JHotDraw redraw mechanism. The calls to FigureChangeListener.figureInvalidated() and FigureChangeListener.figureChanged() show the role of listener objects in the screen redraw procedure.
J3 (S3 How can a problem solution be mapped onto the functionality provided by the software system?)

It is apparent from Figure 4.4 that AbstractFigure.basicMoveBy() is responsible for actually changing the location of the Figure object. The sequence diagram generated by Together for EllipseFigure.basicMoveBy() is shown in Figure 4.5; the choice of EllipseFigure was arbitrary – any one of a number of the concrete subclasses of AbstractFigure would serve to illustrate this interaction. The call to Rectangle.translate() in Figure 4.5 reveals that JHotDraw uses Rectangle objects (termed ‘display boxes’) to represent the positions of Figure objects in the diagram.

![Sequence Diagram](image)

Figure 4.5. The sequence diagram generated by Together for the EllipseFigure.basicMoveBy() method.

J4 (S4 Where is the functionality required to implement a solution located in the software system?)

The class diagram shown in Figure 4.6 reveals that the Figure interface contains a displayBox() method, which is a likely candidate to return a Figure’s display box. Examining the class diagram containing the Rectangle class (see Figure 4.7) reveals the Rectangle.intersects() method, which can be used to test whether two Rectangle objects intersect one another.
Figure 4.6. The class diagram generated by Together for the Figure interface.

```java
Cloneable
Serializable

class CH.java.draw.framework.Figure

+ moveBy(dx:int, dy:int): void
+ basicDisplayBox(origin:Point, corner:Point): void
+ displayBox(origin:Point, corner:Point): Rectangle
+ draw(g:Graphics): void
+ handles(): Vector
+ size(): Dimension
+ center(): Point
+ isEmpty(): boolean
+ figures(): FigureEnumeration
+ findFigureInside(x:int, y:int): Figure
+ containsPoint(x:int, y:int): boolean
+ clone(): Object
+ displayBox(r:Rectangle): void
+ includes(f:Figure): boolean
+ decompose(): FigureEnumeration
+ addToContainer(c:FigureChanges): void
+ addChangeListener(l:FigureChangeListener): void
+ removeChangeListener(l:FigureChangeListener): void
+ invalidate(): void
+ willChange(): void
+ changed(): void
+ canConnect(): boolean
+ connector(x:int, y:int): Connector
+ connectorVisibility(isVisible): boolean
+ connectionInsets(): Insets
+ connectedTextLocation(text:Figure): void
+ getAttribute(name:String): Object
+ setAttribute(name:String, value:Obj
```
Figure 4.7 The class diagram generated by Together for the Rectangle class.
J5 (S5 What alternative functionalities are available in the software system to implement a solution?)

Figure 4.4 illustrates the AbstractFigure.MoveBy(int, int) method, which can be used to move a Figure without changing its dimensions. Figure 4.6 reveals that the Figure interface also contains the methods displayBox(Point, Point) (see Figure 4.8), and displayBox(Rectangle) (see Figure 4.9), which can be used to move and resize a Figure object in one operation. Both methods achieve this by changing the Figure object’s display box directly using the Figure.basicDisplayBox() method; the sequence diagram generated by Together for EllipseFigure.basicDisplayBox() is shown in Figure 4.10.

Figure 4.8. The sequence diagram generated by Together for the AbstractFigure.displayBox(Point, Point) method.
Figure 4.9. The sequence diagram generated by Together for the AbstractFigure.displayBox(Rectangle) method.
Figure 4.10. The sequence diagram generated by Together for the EllipseFigure.basicDisplayBox() method.

\( J6 \) (S6 How does the state of an object change during an interaction?)

Together does not support the extraction of runtime information, so it is not possible to investigate this.

4.3.3 Summary

Together was successful in producing a model of the static structure of the system in the form of a class diagram. Its statically derived interaction diagrams could be used to give an approximation of the runtime behaviour of a single method. There is no functionality for identifying high-level structural components or interactions, save for what can be determined by the analyst from the class and interaction diagrams. Behavioural and design patterns are not automatically identified. The lack of runtime information makes it impossible to measure the load on system components. There is no way to identify hotspots automatically. Some idea of change impact analysis can be obtained using the ‘Search for Usages’ function, which identifies all code locations where an attribute, method, class, interface, or package is used.

Together coped well with the small-scale questions J1-J5: it was able to answer the questions on object collaboration, control structure, mapping, and functionality identification. However, Together’s lack of dynamically-extracted information prevents it from observing changes to the state of an object at runtime.

The strengths of Together were seen as:
- the comprehensiveness of its diagrams due to their generation from source code; and
- its ‘Search for Usages’ functionality.
Together’s principal weaknesses are attributable to its lack of dynamically-extracted information:

- while the diagrams are broad in scope they lack depth;
- it is impossible to focus the diagrams on a particular part of the system’s execution;
- it is difficult to know which are the ‘interesting’ methods for which the analyst should create sequence diagrams;
- sequence diagram generation can be time-consuming;
- references to (methods of) interfaces and abstract classes cannot be resolved to objects, as the implementing/extend extending class cannot be determined statically;
- references to subtypes cannot always be fully resolved, as it is not possible to determine statically whether an object is an instance of the supertype or of one of its subtypes. For example, a reference in a statically derived sequence diagram may be to an object of type Figure, which could resolve to an object of type EllipseFigure at runtime; and
- the inability to examine internal object state.

4.4 Jinsight

4.4.1 Large-scale questions

L1 What is the static structure of the software system?
Jinsight does not display static information.

L2 What interactions occur between objects at runtime?
The execution view can be used to examine object interactions. Part of the execution view for the following execution of the orrery application is shown in Figure 4.11.

Load Orbit
Create a planet
Create another planet
Connect one planet to the other
Adjust the mass of the orbited planet
Start the animation
Stop the animation
Exit Orbit
Figure 4.11. Part of the Jinsight execution view for the orrery application. The coloured horizontal lines represent method calls.

The execution trace (Trace 1) consisted of 1,931,503 events occupying 36,138,055 bytes. Jinsight took 45 seconds to load the trace, with the java.* and sun.* packages being excluded. The reference pattern view (see Figure 4.12) and object histogram (see Figure 4.16) can be used to explore object references.
Figure 4.12. A Jinsight reference pattern view for part of the orrery application.

L3 What is the high-level structure/architecture of the software system? Jinsight does not support architectural extraction.
L4 How do the high-level components of the software system interact?

*Jinsight* does not support the display of interactions between high-level components.

L5 What patterns of repeated behaviour occur at runtime?
The execution pattern view can be used to display repetitions in the execution trace. Figure 4.13 shows part of the execution view for the orrery application with repetition detection turned off. Figure 4.14 shows the same part of the execution view with repetition detection turned on (the execution pattern view).

![Figure 4.13](image)

Figure 4.13. Part of the *Jinsight* execution view for the orrery application with repetition detection turned off.
L6 What is the load on each component of the software system at runtime?

A general overview of time–consuming methods can be gained from the execution view – long bars with white space to the right indicate methods that are active on the stack for a long time. The object histogram can be used to highlight time-consuming objects. Figure 4.15 shows the number of calls to methods of each object, while Figure 4.16 shows the active memory size of the objects.
Figure 4.15. Part of the *Jinsight* object histogram for the orrery application, showing the number of calls to methods of each object. The scale is shown at the top of the window, with black being the lowest and red the highest. Filled rectangles represent objects; outline rectangles represent garbage-collected objects. Diamonds represent the class object of a class.

Figure 4.16. Part of the *Jinsight* object histogram for the orrery application, showing the active memory size of the objects.
L7 What design patterns are present in the software system's implementation?
*Jinsight* does not support the detection of design patterns.

L8 Where in the software system are the hotspots where additional functionality can be added?
*Jinsight* does not support the identification of hotspots.

L9 What impact will a change made to the software system have on the rest of the software system?
The method histogram can be used to show the callers and callees of a method. Figure 4.17 shows the methods called by the selected method (highlighted in yellow). Selecting a method in the method histogram will highlight all uses of that method in the execution view. The invocation browser (Figure 4.18) can be used to show which methods call the selected method and when.

![Method Histogram](image)

*Figure 4.17. Part of the Jinsight method histogram for the orrery application, showing the methods called by the selected method.*
4.4.2 Small-scale questions

In order to address the small-scale questions, a new, more focussed execution trace was created from the following execution.

Start Orbit
Create a planet
Start tracing
Move planet
Stop tracing
Exit Orbit

The execution trace (Trace 2) consisted of 754,431 events occupying 14,462,380 bytes. \textit{Jinsight} took 20 seconds to load the trace, with the \texttt{java.*} and \texttt{sun.*} packages being excluded.

\textit{JI (S1 What are the collaborations between the objects involved in an interaction?)}

The execution view revealed that the \texttt{AWT-EventQueue-0} thread contained most of the ‘interesting’ events. Inspecting this thread revealed the portion of the execution where the redraw methods were called; the relevant part of the execution view is shown in Figure 4.19. Selecting the \texttt{invalidate()} method allowed the call tree shown in Figure 4.20 to be produced, which illustrates the sequence of method calls involved in a screen redraw.
Figure 4.19. Part of the *Jinsight* execution view for the second orrery event trace, showing the redraw methods.

Figure 4.20. Part of the *Jinsight* call tree for the second orrery event trace, showing the call tree from the `invalidate()` method.
Figure 4.21 shows part of the execution view pertaining to the redraw mechanism; Figure 4.22 shows the call tree. The calls to the `FigureChangeListener.figureInvalidated()`, `FigureChangeListener.figureChanged()`, and `FigureChangeEvent.<init>` listener methods illustrate the implicit control structure of the redraw mechanism.

Figure 4.21. Part of the Jinsight execution view for the second orrery event trace, showing the implicit control structure of the JHotDraw screen redraw mechanism.
Figure 4.22. The *Jinsight* call tree showing the implicit control structure of the *JHotDraw* screen redraw mechanism.

**J3 (S3 How can a problem solution be mapped onto the functionality provided by the software system?)**

It was difficult to locate the relevant part of the trace in the execution view, despite tracing only being turned on for a small part of the execution. Figure 4.23 shows a part of the execution view with *EllipseFigure.basicMoveBy()* selected. The call tree for this method, shown in Figure 4.24, reveals that this method calls *Rectangle.translate()* which suggests that the *Figure* object’s position is defined by a *Rectangle* object.
Figure 4.23. Part of the Jinsight execution view for the second orrery event trace, with the EllipseFigure.basicMoveBy() method selected.

Figure 4.24. The Jinsight call tree for the EllipseFigure.basicMoveBy() method.
J4 (S4 Where is the functionality required to implement a solution located in the software system?)

It was difficult to determine where the required functionality was located, as Jinsight does not provide a static view of the software system (e.g. a class diagram). The desired functionality would only be evident in the trace if it (or a related or similar concept) were already implemented.

J5 (S5 What alternative functionalities are available in the software system to implement a solution?)

Again, the lack of a static view showing all of the functionality available in the software system made it difficult to identify alternative functionalities. This is a disadvantage of tools that rely solely on dynamically extracted information and can hence only illustrate the selection of the system’s behaviour that is exercised in the execution. The execution view shown in Figure 4.25 shows that AbstractFigure.displayBox(Point, Point) calls MyEllipseFigure.basicDisplayBox(Point, Point), both of which could be used to change both the position and dimensions of a Figure object in a single operation. However, Jinsight did not identify the general case of Figure.displayBox(Point, Point) and Figure.displayBox(Rectangle), both of which call Figure.basicDisplayBox(). The statically extracted diagrams of Together did identify these methods in the Figure interface. The AbstractFigure.MoveBy(int, int) method identified by Together that can be used to move a Figure object without changing its dimensions was also not evident in the Jinsight diagrams.
Figure 4.25. Part of the Jinsight execution view for the second orrery event trace, which shows that 
AbstractFigure.displayBox(Point, Point) calls 
MyEllipseFigure.basicDisplayBox(Point, Point).

**J6 (S6 How does the state of an object change during an interaction?)**

Jinsight does not support querying the internal state of an object. Memory hotspots (areas of high activity) and object creation and garbage collection can be examined, while references to or from particular objects can be explored using the reference pattern and object histogram.

### 4.4.3 Summary

Jinsight was not able to give information on the static structure or high-level architecture of the system. It provides an array of diagrams for examining dynamic behaviour, but cannot display behavioural information for high-level components. The execution pattern view was used to identify patterns of repeated behaviour. The execution view and object histogram can be used to identify high-activity classes and methods. Jinsight does not support the identification of design patterns or hotspots for extension. The method histogram and invocation browser can be used in conjunction with the execution view to identify where methods are used, which would be useful for change impact analysis.

Jinsight was able to answer questions on object collaboration and control structure. The size of the diagrams made it difficult to identify how a solution could be mapped onto the framework. The lack of a static view hindered the identification of framework functionality. Jinsight does not support analysis of objects’ internal state.
The strengths of *Jinsight* were considered to be:

- a variety of dynamic views;
- accuracy of its diagrams due to dynamically-extracted information; and
- automatic behavioural pattern identification.

The weaknesses of *Jinsight* were seen as:

- difficulty in focussing the visualisation due to the size of the diagrams;
- lack of a static representation of the software system;
- lack of generality in its diagrams resulting from a lack of statically-extracted information; and
- the inability to examine internal object state.

### 4.5 Reflexion models

#### 4.5.1 *jRMTool*

The initial high level model created by the analyst and input to *jRMTool* is shown in Figure 4.26. The reflexion model computed by *jRMTool* is shown in Figure 4.27. *jRMTool* took four seconds to generate the reflexion model. The reflexion model contained 22 nodes and 71 arcs, consisting of 15 convergences (agreement with high level model), 38 divergences (found in the system, but not present in the high level model), and 18 absences (present in the high level model, but not found in the system).

![Figure 4.26. The initial high-level model input to *jRMTool*. Ovals represent high-level system components. Directed arcs represent communication.](image-url)
4.5.2 AVID

The mapping from source entities to high-level entities was identical to that used in Section 4.5.1. The execution trace used was Trace 2 from Section 4.4, as follows.

Start Orbit  
Create a planet  
Start tracing  
Move planet  
Stop tracing  
Exit Orbit  

The execution trace consisted of 754,431 events occupying 14,462,380 bytes. AVID took 15 minutes to load the trace and produce the diagram, with the java.*, javax.*, and sun.* packages being excluded. Figure 4.28 shows the cel view at the start of the execution. Figure 4.29 shows the cel view partway through the execution; the red directed hyperarc represents the call stack at the end of the displayed execution interval. Figure 4.30 shows the summary view of the execution; the black directed arcs represent communication between the high-level entities. The grey rectangles represent high-level system components. The histograms illustrate object population.
Figure 4.28. The AVID cell view at the start of the execution.

Figure 4.29. The AVID cell view partway through the execution.
4.5.3 Large-scale questions

L1 What is the static structure of the software system?
Reflexion models do not show the basic static structure of the software system: they represent a higher level of abstraction. In a degenerate case where each high level entity in the model consisted of one class, the static structure could be shown. However, the reflexion model would be likely to be prohibitively large.

L2 What interactions occur between objects at runtime?
Again, this question is at too low a level to be represented in a reflexion model. In the degenerate case described above, object communication would be shown in an aggregated form (i.e. individual method calls would not be shown).

L3 What is the high-level structure/architecture of the software system?
This is included in the reflexion models in Figures 4.26 – 4.29. The reflexion model technique is reliant on the analyst to specify the high level entities and which source code elements belong to which high level entities.

L4 How do the high-level components of the software system interact?
The arcs in the reflexion models illustrate communication between the objects of the high-level entities. The reflexion model shown in Figure 4.27 was generated using statically extracted information. Although jRMTool can generate reflexion models
using dynamically extracted information, a facility to convert the output of a Java execution trace generator into the format required by \texttt{jRMTool} was not available.

Unlike the reflexion model generated by \texttt{jRMTool}, the model shown in Figures 4.27 – 4.29 was generated by \texttt{AVID} using dynamically extracted information from an event trace. The cel views show communication between high-level entities for specific points in the execution. These views can be combined in an animation. The summary view illustrates communications for the entire execution.

\textit{L5 What patterns of repeated behaviour occur at runtime?}
If dynamically extracted information is included in the reflexion model, the arc annotations show the number of times the communication represented by the arc occurred in the execution. If only statically extracted information is included, a more general representation would result.

\textit{L6 What is the load on each component of the software system at runtime?}
If dynamically extracted information is included, the number of arcs going to/from a node and the numbers on the arcs would give an idea of the relative degree of runtime activity for each high level entity in the model. In the \texttt{AVID} model, histograms illustrate the object population for each high level entity.

\textit{L7 What design patterns are present in the software system's implementation?}
Most design patterns are identifiable only at a level of abstraction below that provided by the reflexion model technique.

\textit{L8 Where in the software system are the hotspots where additional functionality can be added?}
Again, identification of hotspots typically requires a lower level of abstraction than available in a reflexion model.

\textit{L9 What impact will a change made to the software system have on the rest of the software system?}
The reflexion model shows the dependencies that exist between high-level entities. Change impact could be explored by changing the high level entities to which source code entities correspond, removing source code entities, or adding new source code entities to the model.

\textbf{4.5.4 Small-scale questions}

\textit{J1 (S1 What are the collaborations between the objects involved in an interaction?)}
This question is at too low a level of abstraction to be answered using a reflexion model. Even in the degenerate case described above where each high level entity consisted of only one class, individual method calls would not be shown.
J2 (S2 What is the control structure in an interaction?)
This question is also at too low a level of abstraction for the reflexion model technique to be useful.

J3 (S3 How can a problem solution be mapped onto the functionality provided by the software system?)
Reflexion models are at too high a level of abstraction to answer this question. However, the reflexion model would be very useful for validating the analyst’s view of the system in higher-level mapping problems. In this case, the high level model would be refined until it was sufficiently accurate.

J4 (S4 Where is the functionality required to implement a solution located in the software system?)
This question is at too low a level of abstraction for the reflexion model technique to answer.

J5 (S5 What alternative functionalities are available in the software system to implement a solution?)
This question is also at too low a level of abstraction for the reflexion model technique to answer. Alternative functionalities are not evident from the reflexion model.

J6 (S6 How does the state of an object change during an interaction?)
This question is again at too low a level of abstraction for the reflexion model technique. Reflexion models do not show individual objects. Although, unlike jRMTool, AVID shows the progression of the execution, object state cannot be examined. Changes in object population can be observed from the AVID histograms.

4.5.5 Summary
Reflexion models are at too high a level of abstraction to show basic static structure or object interactions. The architecture and high-level interactions were clearly shown in the reflexion model. Only a very general, aggregated impression of patterns of repeated behaviour and runtime load were evident in the reflexion model. The identification of design patterns and extension hotspots were both below the level of abstraction provided by the reflexion model. Change impact can be investigated by altering the input high-level model or the mapping from source to high-level entities.

Reflexion models are at too high a level of abstraction to illustrate object collaborations, control flow, alternative functionalities, or object state. They would be useful for mapping problems at a higher level of abstraction.

The strengths of the reflexion model technique are:
- it illustrates the software system architecture;
- it illustrates the high-level interactions in the system; and
- it enables the analyst to validate their model of the system.
The weaknesses of reflexion models were felt to be:

- the reflexion model technique relies on the analyst to provide an adequately accurate high-level model as input; and
- reflexion models are at too high a level of abstraction for them to answer small-scale questions, such as those relating to object interactions or internal state.

4.6 Together debugger

The execution was that the same as that used to generate Trace 2 in Section 4.4, as follows.

Start Orbit
Create a planet
Start tracing
Move planet
Stop tracing
Exit Orbit

By default, the debugger excludes messages relating to the com.sun.*, java.*, javax.*, org.omg.*, and sun.* packages from its output.

4.6.1 Large-scale questions

L1 What is the static structure of the software system?
Debuggers do not provide static information.

L2 What interactions occur between objects at runtime?
Setting a class breakpoint at every class would show inter-object communication. Setting a method breakpoint at every method of every class would also show the methods involved in the communication, but it would be more time-consuming to set the breakpoints.

L3 What is the high-level structure/architecture of the software system?
This question is at too high a level of abstraction to be answered by a debugger.

L4 How do the high-level components of the software system interact?
Again, this question is at too high a level of abstraction to be answered by a debugger.

L5 What patterns of repeated behaviour occur at runtime?
This could be examined manually by examining the series of breakpoints that are encountered. There is no automatic support for pattern extraction.
L6 What is the load on each component of the software system at runtime?
Class or method level breakpoints would show object or method accesses. No aggregate information is available.

L7 What design patterns are present in the software system's implementation?
This question is at too high a level of abstraction to be answered by a debugger.

L8 Where in the software system are the hotspots where additional functionality can be added?
Again, this question is at too high a level of abstraction to be answered by a debugger.

L9 What impact will a change made to the software system have on the rest of the software system?
This could be assessed to some extent by setting breakpoints on the classes and/or methods involved in the change, and comparing the debugger output before and after the change.

4.6.2 Small-scale questions

J1 (S1 What are the collaborations between the objects involved in an interaction?)
Method breakpoints were set at the following likely methods.

CH.ifa.draw.framework.Figure.moveBy(int, int)
CH.ifa.draw.framework.Figure.basicDisplayBox(Point, Point)
CH.ifa.draw.framework.Figure.displayBox(Point, Point)
CH.ifa.draw.framework.Figure.displayBox()
CH.ifa.draw.framework.Figure.draw(Graphics)
CH.ifa.draw.framework.Figure.displayBox(Rectangle)
CH.ifa.draw.framework.Figure.listener()
CH.ifa.draw.framework.Figure.invalidate()
CH.ifa.draw.framework.Figure.willChange()
CH.ifa.draw.framework.Figure.changed()
CH.ifa.draw.standard.AbstractFigure.moveBy(int, int)
CH.ifa.draw.standard.AbstractFigure.basicMoveBy(int, int)
CH.ifa.draw.standard.AbstractFigure.displayBox(Point, Point)
CH.ifa.draw.standard.AbstractFigure.basicDisplayBox(Point, Point)
CH.ifa.draw.standard.AbstractFigure.displayBox()
The debugger output in Figure 4.31 shows the collaborations involved in the redraw mechanism.
Figure 4.31. Together debugger output showing the method calls involved in a screen redraw in *JHotDraw*.

**J2 (S2 What is the control structure in an interaction?)**

Method breakpoints were set as above, with the addition of a number of likely listener methods, as follows.

```java
FigureChangeEvent.<clinit>()
FigureChangeEvent.<init>()
FigureChangeEvent.FigureChangeEvent(Figure, Rectangle)
FigureChangeEvent.FigureChangeEvent(Figure)
FigureChangeEvent.getFigure()
FigureChangeEvent.getInvalidatedRectangle()
FigureChangeListener.figureInvalidated(FigureChangeEvent)
FigureChangeListener.figureInvalidated(FigureChangeEvent)
FigureChangeListener.figureRequestUpdate(FigureChangeEvent)
```

The calls to *FigureChangeEvent.<init>* in Figure 4.32 illustrate the implicit control structure.
J3 (S3 How can a problem solution be mapped onto the functionality provided by the software system?)

Figure 4.31 shows that AbstractFigure.displayBox() is called when a Figure object is moved. However, it is not evident from this output that a Rectangle object is involved. This highlights the difficulty of knowing where to set breakpoints – a breakpoint at Rectangle.translate() would be required to show the participation of Rectangle (see Figure 4.5).
**J4 (S4 Where is the functionality required to implement a solution located in the software system?)**

The lack of an overall view of the system makes it difficult to find where functionality is located. It could be hypothesised from Figure 4.31 that `Figure.displayBox()` returns the Figure's display box (examining the source code would confirm this), but there is no way to find the `Rectangle.intersects(Rectangle)` method that is required to test for intersection (see Figure 4.7).

**J5 (S5 What alternative functionalities are available in the software system to implement a solution?)**

The key disadvantage of dynamically extracted information is that it only illustrates the system behaviour during one execution of the program. Thus, alternatives are not always apparent, as they would be in static analysis. Additionally, the *Together* debugger output shows only the method names, not the full signature including arguments, so it is not possible to determine which version of an overloaded method is being called (e.g. it is not possible to distinguish between `Figure.displayBox(Point, Point)` and `Figure.displayBox(Rectangle)`).

**J6 (S6 How does the state of an object change during an interaction?)**

Watches were set on four attributes of the `Orbit.MassEllipseFigure` class. The `DEFAULT_MASS`, `MINIMUM_MASS`, and `MAXIMUM_MASS` attributes are defined as static and hence their values should not change. `mass` is an instance variable whose value is expected to change during the execution. A line breakpoint was set at `Orbit.MassEllipseFigure:34`, which is the line “`mass = newMass;`” in the `setMass(double newMass)` method. As expected, the values of the three static attributes do not change, while the value of `mass` changes every time the user of the simulation changes the mass of a planet, which are represented by `MassEllipseFigure` objects. The *Together* debugger interface showing the watches is shown in Figure 4.33.
4.6.3 Summary

Although static information is not shown, dynamic information can be output by setting breakpoints at ‘interesting’ methods or classes. High-level structural and behavioural information is above the low level of abstraction provided by the debugger. There is no functionality to detect repeated patterns of execution, or to show runtime component load. Questions relating to design patterns and extension hotspots are at too high a level of abstraction to be answered using a debugger. Basic change impact analysis can be performed by comparing the output from executions before and after the change.

If breakpoints can be accuracy placed at ‘interesting’ methods, questions about object collaborations and control structure can be answered straightforwardly. The lack of a view of the whole system makes mapping problems and identifying functionality difficult. The dynamically extracted nature of the information means that alternative functionalities are not always apparent, and the lack of full method signatures make method identification confusing. The debugger was able to query internal object state conveniently.

The strengths of the Together debugger are as follows:

- the low level of abstraction would be useful for finding code-level errors;
- dynamically extracted information gives precise output;
• integration with source code makes setting and monitoring breakpoints and
  watches more convenient;
• diagram animation during debugging assists comprehension; and
• the ability to examine internal object state.

The weaknesses of the debugger were found to be:
• the low level of abstraction makes it impossible to answer many higher-level
  questions, such as those relating to the system architecture;
• lack of statically extracted information means only a subset of possible
  behaviour is shown;
• unlike some other debuggers, such as jdb, the Together debugger requires
  source code, which may not always be available, particularly for legacy
  systems;
• it can be very time-consuming to set each breakpoint manually. To obtain
  information comparable to that provided by a tracing tool, method breakpoints
  would be required at every method in the system; and
• it is often difficult to know where to set breakpoints. Setting breakpoints for
  every method would result in information overload.

4.7 Case study summary

A summary comparison of the five dynamic visualisation tools evaluated in the case
study is given in Table 4.1 and Table 4.2. Table 4.1 evaluates the performance of each
tool on the question set; Table 4.2 assesses the performance of the tools on each
question. These comparisons assess each tool’s performance in each task simply as
yes/no: if a tool performed a task sufficiently well, it received a ‘yes’, otherwise a
‘no’.

It is clear from Table 4.1 that Together diagrams and Jinsight were able to answer the
most questions (53%), whereas jRMTool and AVID could answer the fewest (20%).
Comparing tools of similar abstraction levels that use different extraction techniques
indicates that the choice of statically or dynamically extracted information does not
affect significantly the number of questions the tool can answer. This was surprising,
although a larger case study involving more tools would be required before any strong
conclusions could be drawn from this result. Table 4.1 also shows that the reflexion
model technique is unsuitable for small-scale questions whether statically or
dynamically extracted information is used. It would be interesting to assess in this
way the performance of a tool that combines both types of information, such as
Shimba (see Section 3.10) [Systä 2001]. With an abstraction level of 2-4, Shimba
addresses a wider range of abstraction levels than any of tools in this case study. This
range of abstraction levels, combined with the inclusion of both statically and
dynamically extracted information, should allow Shimba to perform well in both the
large- and small-scale questions. Shimba would be expected to be useful in answering
a higher proportion of questions than the tools considered in this case study. 
Unfortunately, Shimba was not available for evaluation.

Table 4.1 reveals that an abstraction level of around 2-3 is optimal in terms of
answering the most questions. Moving away from this point, for small-scale
questions, the tools become less effective as their abstraction levels move towards the
higher (macroscopic) end of the scale, while for large-scale questions the opposite is true. As expected, tools that employ abstraction as an analysis technique were able to answer more large-scale questions than the tool that did not (Together debugger). However, increasing the level of abstraction still further resulted in worse performance in small-scale questions than if no abstraction were used.

A larger case study involving more tools is required before further conclusions can be drawn regarding the effectiveness of the presentation techniques, analysis techniques (other than abstraction), or dynamic extraction techniques.
Table 4.1 Tools summary comparison

<table>
<thead>
<tr>
<th>Tool</th>
<th>Extraction technique (Section 2.4)</th>
<th>Analysis technique (Section 2.5)</th>
<th>Presentation technique (Section 2.6)</th>
<th>Abstraction level (Section 3.1.2)</th>
<th>Large-scale performance (/9)</th>
<th>Small-scale performance (/6)</th>
<th>Overall performance (/15)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Together diagrams</td>
<td>Static</td>
<td>Abstraction</td>
<td>UML diagrams</td>
<td>2-3</td>
<td>3 {L1, L2, L9}</td>
<td>5 {J1, J2, J3, J4, J5}</td>
<td>8</td>
</tr>
<tr>
<td>Jinsight</td>
<td>Dynamic (profiler)</td>
<td>Pattern recognition, abstraction</td>
<td>MSC-based</td>
<td>2-3</td>
<td>4 {L2, L5, L6, L9}</td>
<td>4 {J1, J2, J3, J5}</td>
<td>8</td>
</tr>
<tr>
<td>jRMTool</td>
<td>Static</td>
<td>Abstraction</td>
<td>Graph-based</td>
<td>4</td>
<td>3 {L3, L4, L9}</td>
<td>0 {}</td>
<td>3</td>
</tr>
<tr>
<td>AVID</td>
<td>Dynamic (profiler)</td>
<td>Abstraction, suspension</td>
<td>Graph-based</td>
<td>4</td>
<td>3 {L3, L4, L9}</td>
<td>0 {}</td>
<td>3</td>
</tr>
<tr>
<td>Together debugger</td>
<td>Dynamic (debugger)</td>
<td>Selective instrumentation, suspension</td>
<td>Textual</td>
<td>1</td>
<td>1 {L2}</td>
<td>3 {J1, J2, J6}</td>
<td>4</td>
</tr>
</tbody>
</table>

3 ‘Suspension’ refers to the ability to suspend and resume tracing.
Table 4.2 Questions summary comparison

<table>
<thead>
<tr>
<th>Large-scale questions</th>
<th>Small-scale questions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Question</strong></td>
<td><strong>Success (/5)</strong></td>
</tr>
<tr>
<td>L1</td>
<td>1</td>
</tr>
<tr>
<td>L2</td>
<td>3</td>
</tr>
<tr>
<td>L3</td>
<td>2</td>
</tr>
<tr>
<td>L4</td>
<td>2</td>
</tr>
<tr>
<td>L5</td>
<td>1</td>
</tr>
<tr>
<td>L6</td>
<td>1</td>
</tr>
<tr>
<td>L7</td>
<td>0</td>
</tr>
<tr>
<td>L8</td>
<td>0</td>
</tr>
<tr>
<td>L9</td>
<td>4</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>14/45 (31%)</strong></td>
</tr>
<tr>
<td><strong>OVERALL</strong></td>
<td><strong>26/75 (35%)</strong></td>
</tr>
</tbody>
</table>

Table 4.2 shows that the tools were more successful in answering the small-scale questions: 40% compared to 31%. It also shows that, on average, a tool could answer only 35% of the questions. This may imply that a single software comprehension tool may not be adequate for all tasks. Kazman and Carrière [Kazman 1999] posit that this is the case for architectural extraction, and Richner and Ducasse [Richner 2002a] say this with regard to design recovery. However, it may also suggest that tools require a combination of both statically and dynamically extracted information to perform well in all tasks.

No tools were able to answer either of the large-scale questions L7 (What design patterns are present in the software system's implementation?) and L8 (Where in the software system are the hotspots where additional functionality can be added?). Keller et al. [Keller 1999] describe the role of the SPOOL environment in assisting an analyst in locating three design patterns (Template Method [Gamma 1995 pp.325-330], Factory Method [Gamma 1995 pp.107-116], and Bridge [Gamma 1995, pp.151-161]) in C++ code. Tonella and Antoniol [Tonella 1999] describe a technique based on concept analysis [Siff 1997] and illustrate its use in identifying instances of the Adapter pattern [Gamma 1995, pp.139-150]. The work by both Keller et al. and Tonella and Antoniol stresses the role of the human analyst in identifying design patterns. Demeyer [Demeyer 1998] discusses hotspot identification in Smalltalk HotDraw; the technique employed identified a large number of false positives. Schauer et al. [Schauer 1999] describe the use of the SPOOL environment in identifying hotspots in C++ code, and emphasise the importance of the human analyst. However, Codenie et al. [Codenie 1997] contend that building applications by extending framework hotspots is too simplistic an approach for real-world problems. These papers reveal that detecting design patterns and hotspots is a non-trivial task, and one that can benefit from tool support.
5 Conclusions and future research

5.1 Summary and conclusions

This report has discussed the role of software comprehension in general, and dynamic analysis and visualisation in particular, in the wider context of software engineering. Techniques for dynamic analysis and visualisation were discussed. Fourteen dynamic visualisation tools were described and compared, and five of them were evaluated empirically in a case study. This section draws conclusions from this report and presents directions for future research in the field of dynamic visualisation.

As discussed in Section 2.1, the complex interactions typical of object-oriented software systems mean that dynamic analysis is often more appropriate than static analysis for software comprehension tasks. However, dynamic analysis captures only a subset of the possible behaviour of the program. The diagrams produced by dynamic analysis are narrow and deep, while those produced by static analysis are wide and shallow. Figure 5.1 illustrates this comparison. The fchild object is an attribute of initial. In practice, fchild is an instance of one of three Tool subclasses: HandleTracker, DragTracker, or SelectAreaTracker. This statically extracted diagram shows the three possible outcomes of the user clicking on some part of a diagram: on either a handle, a figure, or a blank space, respectively. The diagram cannot show messages that occur after message 1.8 (the call to mouseDown(e, x, y):void), as these depend on the type of fchild, which is known only at runtime and hence cannot be determined statically. The diagram is wide but shallow: it shows all of the possibilities. A dynamically extracted diagram would show one possibility in more detail: it would be narrow but deep.
There are three possibilities for combining the benefits of both static and dynamic information to produce a suitably wide and deep visualisation. Firstly, the analyst can ensure that a representative trace is extracted. This raises the questions of how the analyst can ensure that the trace is representative, and how he knows that it is representative enough for the task at hand. Most dynamic analysis tools implicitly require the analyst to perform this function. Secondly, a tool can combine multiple event traces into a single visualisation; this approach is used in Dali and RMTool. Thirdly, statically and dynamically extracted information can be combined, as in Shimba. The key problem of ensuring a representative trace is inherent in dynamic visualisation, even when one of the latter two techniques is employed.

A number of techniques for extracting, analysing, and presenting dynamic visualisations were described. These techniques were implemented in the tools described in Section 3 and evaluated in Section 4. A principal requirement of any...
dynamic visualisation tool is the ability to focus the visualisation on the events that are of interest to the analyst in the particular context in which he is working.

It is clear from the case study results in Table 1 and Table 2 that no one dynamic visualisation tool can answer all questions that are typical of a software comprehension or reverse engineering effort. Some tasks are less well supported than others, and some tasks are beyond the capabilities of all current tools. This implies that current dynamic visualisation tools are not adequate in isolation for supporting software comprehension, and must be employed along with other software engineering tools if all typical issues are to be addressed. The above results also reveal that the application of dynamic visualisation tools in combination can improve performance. Tools employing higher levels of abstraction were more successful in addressing large-scale questions, while those using a lower level of abstraction were more useful for small-scale question; tools employing an abstraction level of 2-3 were most generally effective. These results also suggest that a combination of statically and dynamically extracted information may improve performance. The visualisations generated from statically extracted data are more general but less precise than those obtained from dynamically extracted data: statically extracted visualisations are wide but shallow, while dynamically extracted visualisations are narrow but deep. The lack of a single dynamic visualisation tool that performs well in all tasks is likely a large contributory factor in the lack of use of dynamic visualisation tools outwith the context of research. Analysts are evidently using alternative types of tool to obtain the information they require for software comprehension.

5.2 Contributions of this report

Section 3.1.1 proposes three criteria for characterising dynamic visualisation tools, namely:

1. The technique used to extract the dynamic information from the software system.
2. The methods of analysis that are applied to the extracted data to improve its comprehensibility and usefulness to the analyst.
3. The way in which the results of the visualisation are presented to the analyst.

These criteria were derived from the properties of extant dynamic visualisation tools, and are hence a suitable way of characterising such tools.

Section 3.1.2 presents a scale on which the level of abstraction at which a software comprehension tool operates can be measured. This scale is illustrated in Figure 3.1. The ordinal values range from 1 (microscopic information) to 5 (macroscopic information). The dynamic visualisation tools described in this report were placed on the scale as shown in Figure 3.2.

Section 3 makes use of a template for categorising dynamic visualisation – and other software engineering – tools. It is based around the five headings of Overview, Description, Evaluation, Comparison, and Assessment. A number of alternative taxonomies have been proposed in the literature, including work by Myers [Myers 1986], Chikofsky and Cross [Chikofsky 1990], Stasko and Patterson [Stasko 1992],
Price et al. [Price 1992, Price 1993], and Roman and Cox [Roman 1993]. Price et al. [Price 1993] propose a detailed, multi-level taxonomy for classifying software visualisation tools. Unlike earlier taxonomies that have derived categorisations based on observations of tools, Price et al. justify their categories (Scope, Content, Form, Method, Interaction, and Effectiveness) based on the theory of visualisation tools. They then attempt to classify a selection of software visualisation tools according to this taxonomy. The software visualisation tools in this report are categorised according to four categories that were observed from the extant dynamic visualisation tools (extraction, analysis, and presentation techniques, and abstraction level). There is some commonality between these categories and those of the taxonomy of Price et al. While this categorisation may be less detailed than the taxonomy of Price et al., it provides much of the cogent information that may be required when selecting a dynamic visualisation tool for a software comprehension or reverse engineering task.

Section 4 presents two sets of questions that can be used in the evaluation of software comprehension tools. The large-scale questions are generic, while the small-scale questions are specified in terms of the specific evaluation being undertaken. Specific small-scale questions for use in a JHotDraw case study are presented. A number of tool evaluation techniques are discussed in the literature, e.g. by Globus and Uselton [Globus 1995], Storey et al. [Storey 1996], Mulholland [Mulholland 1997, Mulholland 1998, Mulholland 1999], Hatch et al. [Hatch 2001], and Knight [Knight 2001]. Storey et al. [Storey 1996] evaluated the usability of three user interfaces to the Rigi reverse engineering tool by observing users completing a set of software maintenance tasks followed by a questionnaire and an interview. This technique is similar to that discussed in this report as it evaluates a tool by assessing its performance in a series of typical tasks. However, Storey et al. used a group of twelve volunteers to evaluate the tool, while the evaluation described in this report was carried out by a single user. The small tasks involved were intended to be typical of those performed by software maintainers working towards a larger goal; a trade-off was necessary between experiment time and task complexity. They were divided into two groups of four tasks, ‘abstract’ and ‘concrete’, which were concerned with high- and low-level understanding, respectively. These task groupings are similar to the large- and small-scale tasks used in this report to typify general software comprehension and specific reverse engineering tasks, respectively, though the tasks of Storey et al. are more simplistic. While useful results were obtained in terms of the relative usabilities of the tool interfaces, the paper concludes by identifying the need for a larger user group, more tasks, longer time, and greater experimental control.

This report presented a detailed survey of the current state of the art in dynamic visualisation tools, and an empirical evaluation of the available tools. Other comparisons in this and related fields include work by Bassil and Keller [Bassil 2001a, Basil 2001b] on software visualisation tools, Sim and Storey [Sim 2000] on program comprehension tools, and Bellay and Gall [Bellay 1997] and Kollmann et al. [Kollmann 2002a] on static reverse engineering tools. Kollmann et al. [Kollmann 2002a] compared the class diagram synthesis facility of Together with three other tools. While basic diagram generation results were broadly similar across the tool set, some associations were not detected by Together. The research tools in the study were able to handle more advanced diagram concepts than the industrial tools.
5.3 Future research directions in dynamic visualisation

Stroulia and Systä [Stroulia 2002] identify three key future research areas for dynamic analysis and visualisation. The first of these is concerned with presenting the model of dynamic behaviour to the end user. As discussed in Section 2.6, there are a variety of techniques for presenting visualisations to the analyst. The important factor is that the model should be presented clearly and accurately. If multiple models are used, the semantic relations between them should be clear. It would also be useful to allow the user to navigate easily between static and dynamic views and between high and low levels of abstraction.

Secondly, in addition to using reverse engineering techniques to analyse legacy systems and other completed programs, there are possible applications in the forward engineering phase of software development. The application of both static and dynamic reverse engineering techniques during development allows the verification of software structure and behaviour, respectively, and can also assist in generating accurate documentation. It would be especially desirable if the reverse engineering tools employed could produce standard OOAD (object-oriented analysis and design) models, such as UML diagrams. This parity between the forward- and reverse-engineering models would also help to support maintenance, reuse, reengineering, and round-trip engineering.

Thirdly, application integration and migration, for example the migration of legacy systems to the web, is another area where there is scope for research. Arnold and De Pauw [Arnold 2003] describe the Websight tool for visualisation of web services.

Müller et al. [Müller 2000] identify a number of issues for reverse engineering research, namely:

- software process and product improvement for reverse engineering;
- integration of forward and reverse engineering;
- tool integration and adoption;
- end user customisable tools;
- industrial evaluation of tools and techniques
- cost and risk assessment and control; and
- software evolution education.

Both Stroulia and Systä [Stroulia 2002] and Müller et al. [Müller 2000] identify closer integration of forward and reverse engineering as a useful topic requiring further research.

A larger case study involving more tools is required before further conclusions can be drawn regarding the effectiveness of the presentation techniques, analysis techniques (other than abstraction), or dynamic extraction techniques. Future work should investigate tools that incorporate both statically and dynamically extracted information and allow the analyst to move conveniently between abstraction levels; such tools would have the potential to address many of the issues identified in this paper. Dynamic visualisation tools that incorporate design pattern and hotspot recovery provide another interesting research prospect.
References


Budd T, A Little Smalltalk, Reading, MA: Addison-Wesley, 1987


Computer Science 1543, pp.82-84, Berlin: Springer-Verlag, 1998


