AR-009-029

DSTO AUSTRALIA

ELECTRONICS RESEARCH LABORATORY

Information Technology Division

RESEARCH NOTE
ERL-0838-RN

A VISUAL APPROACH TO THE ANALYSIS OF LARGE SOFTWARE SYSTEMS

by

Philip Dart, Gina Kingston, Rudi Vernik and Stefan Landherr

SUMMARY

Analysing a large software system is a difficult and time consuming task, even with current automated assistance. Measures taken of software systems may be too abstract, applicable only to a particular class of system or simply too subjective. This paper examines problems associated with using existing tools and measures, and then presents an alternative approach to the analysis of large software systems based on system views that have been designed for a specific analysis task. We detail various abstract and concrete views of static system structure, and describe a software analysis environment developed to experiment with this approach. The concepts reported in this paper were developed during early enabling research undertaken as part of the SMAT task. Many of the concepts have already been included in other components of this task, including the SEE-Ada tool for software evaluation.

JULY 94

APPROVED FOR PUBLIC RELEASE

POSTAL ADDRESS: Director, Electronics Research Laboratory, PO Box 1500, Salisbury, South Australia, 5108.

CONTENTS

Page No.

LIST OF FIGURES

THIS PAGE IS LEFT INTENTIONALLY BLANK

1 Introduction

Software systems are already among the most complex systems constructed by humankind. As the size, complexity and criticality of software systems increases, so too does our reliance on and investment in these systems. This provides strong motivation for an effective means of assessing software systems for quality. Many people involved in the software lifecycle, from users to developers and maintainers, require some form of software assessment. However the properties in which they are interested will vary considerably according to each person's role. Such properties include reliability, portability, maintainability and performance.

No individual can understand all aspects of one of the large and complex software systems in current use. Someone attempting to analyse a complex software system must have a clear goal in mind. This allows them to focus on specific aspects of the system which, under analysis, will reveal the required information about a particular property.

As an analogy, consider an inspector tasked with checking the welds on piping in a nuclear power plant. This person would not have complete knowledge of the structure and workings of the plant and would not need it. The inspector's view of the plant would be of a complex network of welded pipes. Armed with expertise in the processes and properties of welding, the inspector would traverse this network according to some pre-planned scheme carrying out tests chosen specifically for the task. Similarly, someone analysing a software system consisting of millions of lines of code could not possibly expect to gain complete knowledge of the system. The analyst would have to view the system from some abstract perspective that masks out the features not required for their analysis. Such a view could, for example, reveal the structure of the system at a software component level, but hide the details of the components.

Unfortunately, while pipes are concrete objects that can be inspected directly, views of a software system, such as the one described above, exist only as abstractions from the source code. While the source code is (usually) specified in some concrete textual form, such abstract views have not been specified in this way. They need to be mapped onto concrete views to give the software analyst something to actually look at. An abstract view can be mapped onto a number of concrete views, permitting an analyst to use whichever is suited to the task at hand. For example, an abstract view of a software system which included the system structure and the names of components, and which had the form of a tree, could be displayed graphically as a hierarchy of names connected by lines, or textually as a list of names indented to indicate the system structure. These are similar to the welder's views of the nuclear power plant. The abstract view contains the connections between components and the usual concrete views are the pipes and the plans showing the pipes.

Of course this approach, by implication, assumes some form of automated assistance which will extract abstract views of large software systems and display them in suitable concrete forms for the software analyst. This paper describes such an environment, designed to allow us to experiment with various abstract and concrete views in order to assess their value to a software analyst. While the environment has been implemented to work with Ada software systems, the concepts involved readily map to any language with facilities for structuring large systems.

Much work has been done previously in the use of measures to support the analysis of programs [1]. The approach discussed here is consistent with approaches based on measures. Measurement values can be considered as components of abstract views, and these components can be viewed concretely, for example, in a textual form as numbers or in a graphical form as histograms. On the other hand, we see no reason to restrict ourselves to abstract views which consist solely of measures.

Section 2 of this paper briefly examines the limitations of existing approaches to software analysis and introduces our approach. Section 3 considers possibilities for abstract views which relate to software system structure and looks at concrete views and the restrictions placed on them by the requirement for scalability. Section 4 gives observations we have made about this approach. It includes looking at the tool we developed to support this approach and some examples of how it has been used. It also highlights further developments that fit into the framework of this approach.

Finally, while the major aim of this work is to develop an approach to the analysis of large software systems, a secondary aim is to determine what information is most useful in performing the analysis. While we have concentrated on static, structural analysis of the source code of the software system, other

aspects of the specification and design of the system, the problem domain and the process used to develop the system should also be used during analysis. Determining what information would be most useful to the analysis process gives an indication of what information should be provided by the specification and design phases of the software development process.

2 Approaching Analysis

As stated earlier, many people involved in the software lifecycle need to be able to assess software, but the properties in which a person is interested will vary considerably according to that person's role. We will focus on one particular role, that of the software maintainer, to enable some of the issues involved in software analysis to be examined in detail.

Consider the situation in which you, the reader, are in charge of a team that will be responsible for the maintenance of a very large software system which is being developed under contract. This system is expected to have a relatively long lifecycle, during which your team could reasonably be expected to make a variety of modifications, both to fix bugs and to change the system functionality. The source code and documentation for the first build of the system has been delivered to you, providing you with the opportunity to give feedback to the contractor, with the aim of making your system maintenance task easier. What do you need to know about the system to enable you to provide this feedback?

2.1 Using Metrics

One approach you could take would be to use measures or metrics for the delivered system. A measure is *a quantitative assessment of the degree to which a software product or process possesses a given attribute* [2]. Throughout this section we will only be considering measures with numeric values and shall use the terms measure and metric interchangeably.

Metric values could be delivered by the contractor with each build of the system, or (for some metrics) generated from the delivered source code by a variety of available commercial products. They might include a value for 'Maintainability', which, if it reflects the appropriate properties of the delivered system, will give an indication of how difficult the code will be to maintain. The maintainability of your system might be reported as 0.65 on a scale of zero to one. What does this tell you? It tells you that the system probably won't be as easy to maintain as it should be. Unfortunately, this information alone is not sufficient feedback to enable the contractor to modify the system to make it easier to maintain. You need to be able to specify where and what the problems are.

One commercial system available for software assessment is AdaMAT/D™ [3], which can generate the values of about three hundred different metrics for each Ada module. These metrics form a hierarchy, as illustrated in Figure 1. The leaves of the hierarchy, called *data elements*, are direct measures of properties of a software component, for example, the 'Proportion of loops that are named'. The values for data elements are then combined as weighted sums to form higher level metric values. These are also combined, and the process is continued until values for the top level metrics (called level 1 software quality goals) such as Maintainability are derived. The metric values can be generated for each module and a further set can be generated as a summary for the whole system. One of these metrics is Maintainability, and the value of this metric, when taken from the summary, is like that described above. Of course, in this case, there is much more information than a single metric value to work with. In fact a large system can easily have over a thousand files, giving about 300,000 metric values. The volume of this information alone can cause problems. In some of our earlier work [4], we used a relational database system to help browse this large amount of metric data to identify areas of code requiring further examination. Although we had some success with this approach, we found it difficult to relate the identified areas to the software structure.

Unfortunately, the volume of information generated is not the only problem with using this kind of metric analyser. Another problem is that the metrics may be too abstract to be of use. While the lowest level metrics are direct measures of properties of the system, as we move up the metric

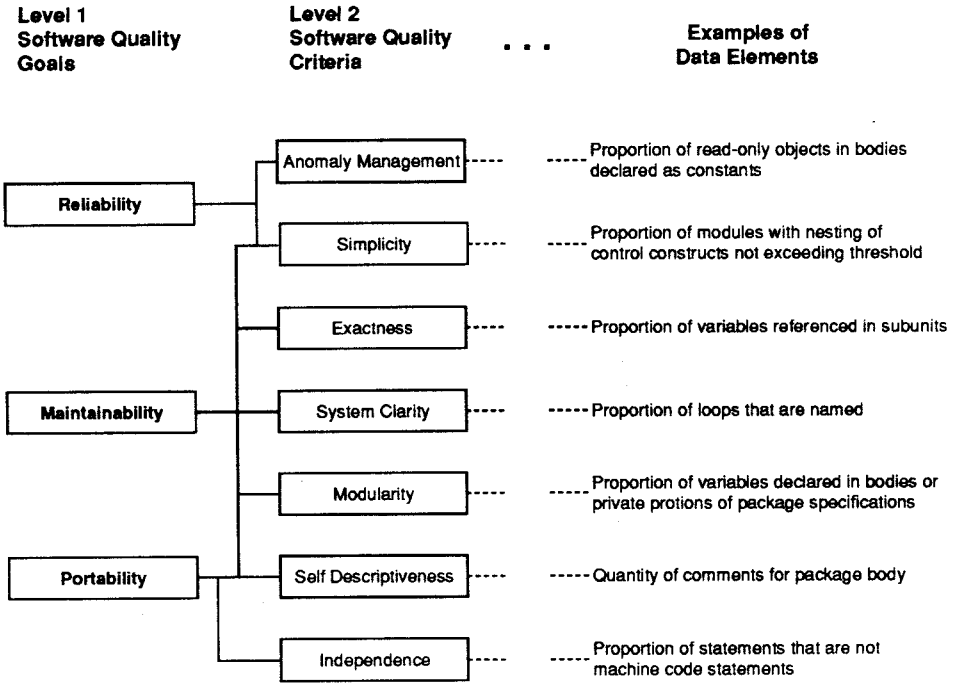


Figure 1 An Extract from the AdaMAT/D Metrics Hierarchy

hierarchy it becomes increasingly difficult to relate the metrics to any such property. Without some definite relationship, the metrics are of little value. A symptom of this problem is that system code can be modified to achieve better metric values without achieving better code. A simple example, but one which is difficult to address, arises from the use of 'Quantity of comments for package body' in the derivation of Maintainability. Blank or nonsense comments can be used to boost up the value for the former metric, thereby increasing the value for the later metric. Our experience with metrics like those provided by AdaMAT/D has resulted in us using the low level metrics but making little or no use of the higher level ones.

Finally, a further problem is that the metrics may be applicable only to a particular class of system and are highly context dependent. The following passage is an extract from the preface to the AdaMAT/D reference manual [3]:

> Scores may be objectionable to some because of their implied grading scheme (i.e. 0.90 – 1.00 is A work, 0.80 – 0.90 is B work, etc). No grading scheme is intended. In fact, it is not generally meaningful to compare scores between metrics. Moreover, comparisons of scores for a particular metric, without appreciating the differing applications, requirements, development environments, and so on may be misleading. Specifically, non-adherences are often justifiable in the context of software requirements, development constraints, processor constraints, implementation environment constraints, etc. . . . Software trade-offs, which are inevitable in the development of software, must be taken into account in the accurate interpretation of AdaMAT scores.

Thus metrics must be interpreted in the context that they were derived and any interpretation will depend on the perspective of the user.

2.2 Viewing Structure

Another approach to maintenance analysis would be to examine the structure of the system to look for potential problem areas. This activity could be supported by a tool that provided structured views of the system.

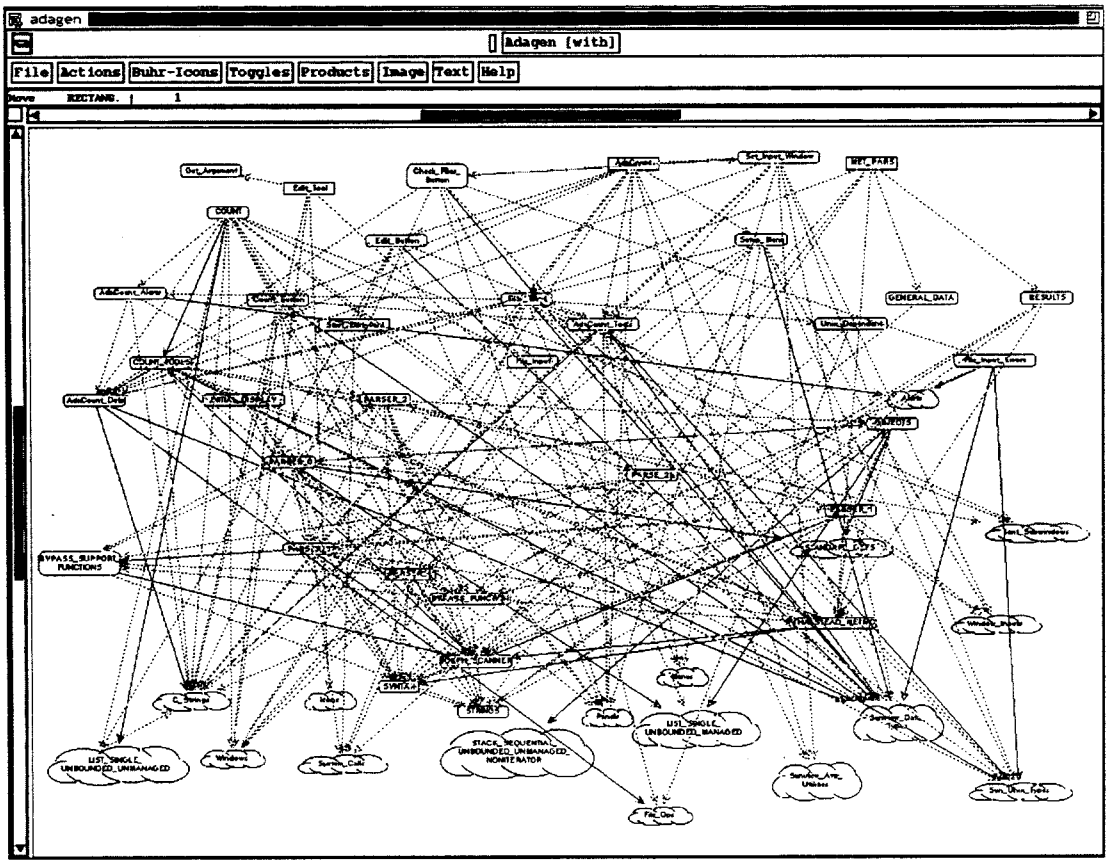


Figure 2 Adagen "**With** Dependency Diagram"

One commercial tool that provides this kind of facility is Adagen™ [5]. An example of an Adagen view of the **with** dependencies in an Ada system is given in Figure 2. The Ada system depicted in this view contains only about fifty components. Even so, the view lacks clarity because it attempts to display the complex structure of the system in its entirety. The example illustrates that this style of diagram quickly becomes incomprehensible as the number and interconnectivity of software components increases. A depiction of a **with** graph that shows all the vertices and edges in this manner is simply not scalable. Since we are interested in analysing systems with potentially thousands of components, some alternative system views are required. For systems of this scale, even displaying the component names may be a problem.

There are many other systems based on graphical notation. Some of these try to reduce the complexity of the graph by limiting the amount of information displayed at any time [6, 7]. However, there is no overall view of the system and useful information is lost.

2.3 Our Approach

Having examined shortfalls in existing approaches, we must now return to the original question: what do you need to know about the system to enable you to provide feedback to the contractor? Answering this question can be split into two steps. First, you need to be able to quickly and easily locate problem areas or *hotspots* in the system, where the definition of a 'hotspot' depends on the kind of analysis being performed. Second, you need to determine the cause of the hotspots and to design modifications to the system that will eliminate or reduce them.

In general, to provide assistance to a software analyst we first need to determine what the analyst thinks of as hotspots and what information about a system is required to find them. Information resulting from this knowledge acquisition process is used to develop *abstract views* that will form the basis for an analysis session. Next we need to apply visualisation techniques to each abstract view to form *concrete views* that make the information available in the manner which best assists

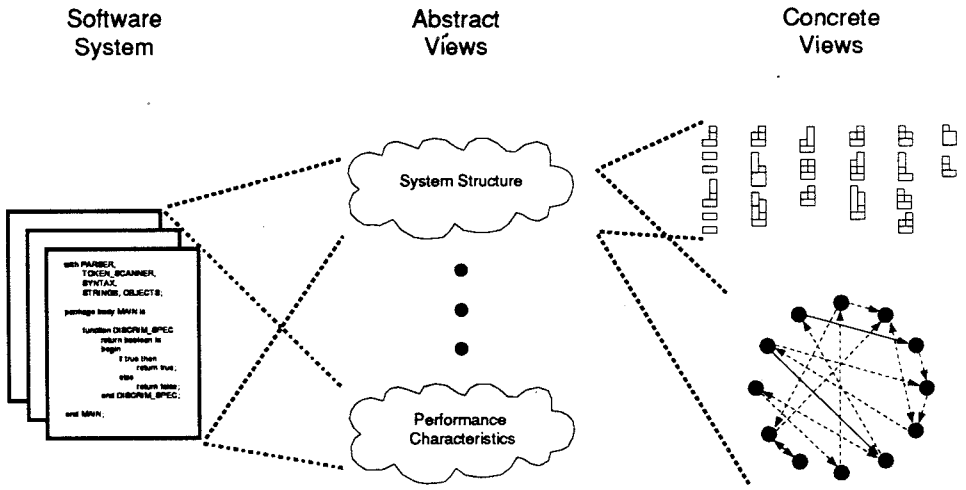


Figure 3 Examples of Abstract and Concrete Views

the analyst to perform the required task. The relationships between a system and the different views are illustrated in Figure 3.

In order to locate maintenance hotspots, we need to know what properties of the code affect maintainability. In its basic form, software maintenance involves modifying software systems [8], so we are interested in properties of the code that make it difficult to modify the system to achieve the desired result. Symptoms of this include: locating which parts of the system need to be modified is difficult; understanding a system component which needs to be modified requires understanding much more of the system; and modifying part of the system has a disproportionate effect on the rest of the system. One way to reduce all of these problems is to limit interdependencies between components of the system, and have, in effect, a system with as simple a structure as possible. Some structure will certainly be necessary to enable the system to support the functionality for which it was designed. On the other hand, you could reasonably expect that for any part of the system which is strongly coupled with the rest of the system, there will be an equally strong justification for that part of the design. Similarly, there would be some structures, acceptable by the rules of the source language, which from a maintenance point of view could have no reasonable design justification.

Thus maintenance hotspots will include parts of the system with high levels of interdependency. Similarly, performance hotspots would include parts of the system in which large percentages of the execution time is spent. Section 3 looks in detail at abstract views of software system structure and related concrete views that can be used to highlight maintenance hotspots.

3 Abstract and Concrete Views of System Structure

As discussed earlier, it is possible to build extremely complex software systems. In order to analyse these systems, high level abstract views of system structure are needed. In this section, we restrict ourselves to examining such abstract views. These views will still be based on source language constructs, in this case for Ada.

3.1 Visibility Graphs for Ada

Ada was designed with the construction of large software systems in mind and so it provides a range of constructs for use in structuring software systems. Some of these are depicted in Figure 4. The following extract from [9] gives an overview of packages, the major structural feature of Ada.

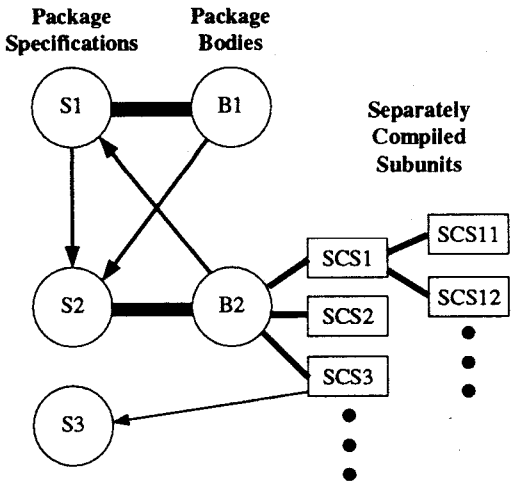


Figure 4 Ada Constructs used to Structure a Software System.

A package is a black box which provides services to users through an interface defined by a package specification. The package specification and the package body are distinct components of the program text in Ada and may be separately compiled if desired. The package body is hidden from the user who may only use the package in the way defined in the specification.

In particular, we will look at views based on *visibility* dependencies, dependencies indicating that objects in the specification of one package are visible in another package. These dependencies, depicted by arrowed lines in Figure 4, are provided through the use of the **with** statement. Dependencies created by using **with** statements are directed, that is, if S1 **with**s S2 then the contents of the specification of S2 are visible inside S1, but not vice versa. While such dependencies do not necessary imply one package *uses* something from the **with**ed package (by analogy, a connected pipe may have nothing flowing in it), each such dependency implies the potential for use.

Consistent with the black box view of packages, and with the approach taken in this paper, we will consider an abstract view in which the separately compiled subunits of a body are hidden within that body. This means that any dependency between a separately compiled subunit and a package (for example SCS3 to S3 in Figure 4) will appear to be between the containing body and the package (B2 to S3). This simplifies the structure depicted in Figure 4 to one containing only package specifications and bodies, and dependencies from bodies or specifications to specifications. Ada permits **with** dependencies that involve components other than packages. For simplicity we treat all such components as packages in our abstract views. Similarly we ignore the effect of the Ada *renaming* construct on visibility in the abstract views.

According to the semantics of the Ada language, declarations visible in the specification of package are also visible in the body of the package, so a dependency from the specification of a package P1 to a package P2 can also be considered as a dependency from the body of P1 to P2. Adding to this Ada's rule that the dependencies between package specifications must not form a cycle, we find that an Ada software system can be viewed as two graphs, both containing vertices representing packages. The first, or *spec-graph*, is a directed acyclic graph whose edges represent dependencies between package specifications. The second, or *body-graph*, is a super-graph of the first, possibly containing cycles, whose edges represent dependencies from package bodies to package specifications. The spec-graph and body-graph for a simple software system are shown in Figure 5. These two graphs represent abstract views of the same software system. The information in these graphs can be extracted from the source code for the system. In more abstract terms, each of these views is a directed graph, or *digraph*, consisting of a set of vertices V and edges E.

Definition Let S be a software system containing packages $\{p_i \mid i = 1, \ldots, N\}$ where each package p_i is represented by a vertex v_i and V is the set of these vertices. Let E_s be the set of

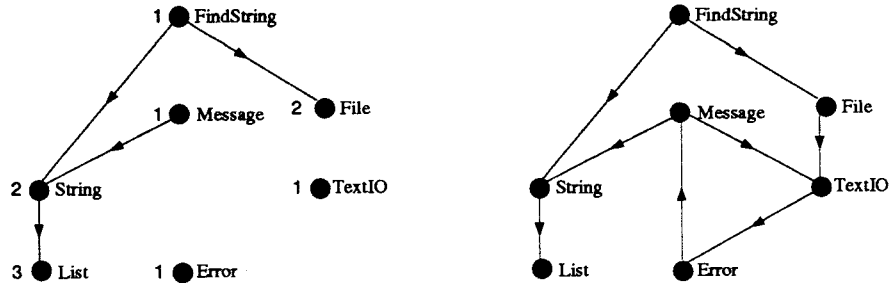


Figure 5 The Spec-Graph and Body-Graph for a Simple Software System.

pairs of vertices (v_i, v_j) for which the specification of p_j is visible in the specification of p_i, in S. Let E_b be the set of pairs of vertices (v_i, v_j) for which the specification of p_j is visible in the body of p_i but not visible in the specification of p_i, in S. Then the **visibility-graph** for S is the triple (V, E_s, E_b). A mapping exists between a visibility-graph (V, E_s, E_b) and two digraphs, a **spec-graph** (V, E_s) and a **body-graph** $(V, E_s \cup E_b)$.

Note that by definition E_s and E_b are disjoint. Throughout this paper, the software system S will always be defined by the context. We also take the liberty of using package names to denote vertices throughout the paper, as illustrated in the following example.

Example The visibility-graph from Figure 5 is given by the following:

$$V = \{\text{Error, File, FindString, List, Message,String, TextIO}\}$$
$$E_s = \{\text{(FindString, File), (FindString, String), (Message, String), (String, List)}\}$$
$$E_b = \{\text{(Message, TextIO), (File, TextIO), (TextIO, Error), (Error, Message)}\}$$

3.2 Levelling

As we illustrated in Section 2.2, a graph view that depicts all the vertices and edges is not scalable. Ideally we want a canonical two-dimensional representation of a graph which is scalable within the limits of current visual display units. In reality, we have to settle for mapping characteristics of the graph on the display attributes in the best way possible. Partially ordering the vertices of a graph permits them to be levelled along one dimension of the display in a manner that directly reflects dependencies within the graph.

Definition Let D be an acyclic digraph (V, E). Then the **level** $level(D, v_i)$ of a vertex $v_i \in V$ is defined as follows:

$$level(D, v_i) = \begin{cases} 1 + max\{level(D, v_j)\} & \text{if } (v_j, v_i) \in E \text{ for some } v_j \\ 1 & \text{otherwise} \end{cases}$$

Example Let $D \equiv (V, E_s)$ be the spec-graph of Figure 5. Each vertex $v_i \in V$ is labelled with $level(D, v_i)$ in the figure.

Using the information in the visibility-graph and the levels of the vertices in the associated spec-graph we can define a concrete view of the system.

Definition The **Spec View** of a system is a concrete view based on the visibility-graph of the system, with associated spec-graph $D = (V, E_s)$. The spec view has the following properties:

- Each vertex $v_i \in V$, corresponding to a package, is represented by a rectangular icon.
- The icons are arranged in columns so that the icon for vertex $v_i \in V$ is in the column given by $level(D, v_i)$.

Spec View **Body View**

Figure 6 A Spec-View and a Body-View for a Simple Software System.

Note that this definition does not allow for the display of package names. This was a conscious choice on our part in designing this concrete view. Displaying names would significantly reduce the number of components that could be viewed, and our experience has shown us that misnamed components can mislead analysts.

From the spec view of a software system, we hope to be able to see if the system's specifications are tightly coupled. If they are then any changes made to a specification in the maintenance phase are very likely to propagate through the system.

Example The spec view of the simple software system with spec-graph given in Figure 5 is shown in Figure 6. The icon in the third column represents the package List which has level 3. In this system most of the packages are at the top level. This means that changes are less likely to propagate through the system. We expect Spec Views to have a tall narrow shape indicating that these types of connections have not been overused. If these have been overused then changes to a specification could result in a large percentage of the code needing recompilation. Note that the order of the packages within the columns is arbitrary.

3.3 Reduced Visibility Graphs and Clusters

Levelling, as defined above, can not be applied to graphs containing cycles. Since body-graphs can contain cycles, a body-graph must be mapped into an acyclic form before levelling can be applied.

Definition Let E be a set of edges. Then **reachable** and **cyclic** are defined as follows:

$$reachable(E, v_i, v_j) \equiv (v_i, v_j) \in E \text{ or } ((v_i, v_k) \in E \text{ and } reachable(E, v_k, v_j) \text{ for some } v_k)$$

$$cyclic(E, v_i, v_j) \equiv reachable(E, v_i, v_j) \text{ and } reachable(E, v_j, v_i)$$

Example For the set of edges E in the body-graph shown in Figure 5, $reachable(E, File, List)$ and $cyclic(E, TextIO, Message)$ both hold.

We eliminate cycles from a body-graph by compressing the vertices in interconnected cycles in the graph into a single vertex in a reduced form of the graph. This corresponds to replacing the strongly connected components of the graph by a single vertex.

Definition The **reduced-visibility-graph** $G^r \equiv (V^r, E_s^r, E_b^r)$ of a visibility-graph $G \equiv (V, E_s, E_b)$ is defined as follows:

$$V^r = \left\{ P_{\mu(j)} | v_i \in V \text{ and } \begin{array}{l} \mu(i) = \mu(j) \text{ if } cyclic(E_s \cup E_b, v_i, v_j) \\ \mu(i) \neq \mu(j) \text{ otherwise} \end{array} \right\}$$

$$E_s^r = \{ (P_{\mu(i)}, P_{\mu(j)}) \mid (v_i, v_j) \in E_s \text{ and } \mu(i) \neq \mu(j) \}$$

$$E_b^r = \{ (P_{\mu(i)}, P_{\mu(j)}) \mid (v_i, v_j) \in E_b \text{ and } \mu(i) \neq \mu(j) \}$$

A mapping exists between a reduced-visibility-graph(V^r, E_s^r, E_b^r) and two digraphs, a **reduced-spec-graph** (V^r, E_s^r) and a **reduced-body-graph** $(V^r, E_s^r \cup E_b^r)$

Example Using sets of packages to denote vertices, the reduced-visibility-graph of the visibility-graph from Figure 5 is given by the following:

$$V^r = \{\{\text{Error, Message, TextIO}\}, \{\text{File}\}, \{\text{FindString}\}, \{\text{List}\}, \{\text{String}\}\}$$

$$E_s^r = \{(\{\text{FindString}\}, \{\text{File}\}), (\{\text{String}\}, \{\text{List}\}),$$
$$(\{\text{FindString}\}, \{\text{String}\}), (\{\text{Error, Message, TextIO}\}, \{\text{String}\})\}$$

$$E_b^r = \{(\{\text{File}\}, \{\text{Error, Message, TextIO}\})\}$$

This visibility graph is depicted in Figure 7. As the reduced-body graph is acyclic, the level function can be applied it. The levels for the reduced-body-graph are also shown in Figure 7.

Definition The **Body View** of a system is a concrete view based on the reduced-visibility-graph,$G^r = (V^r, E_s^r, E_b^r)$, of the system. The body view has the following properties.

- Each vertex $v_i \in V^r$, corresponding to a group of packages, P_i, is represented by a rectangular icon. The size of the icon indicates the number of packages in P_i.
- The icons are arranged in columns so the icon for vertex $v_i \in V^r$ is in the column given by $level(D, v_i)$, where $D = (V^r, E_s^r \cup E_b^r)$.

From the body view we can quickly identify any groups of packages. We also expect to see more levels than in the spec view. If this does not happen, then **withs** in the specifications have probably been overused. Normally you would only expect to have one package at the first level in a body view. More than one package could indicate that redundant packages are being considered, or that the system has multiple control threads. Occasionally other columns contain only one package. In some cases this corresponds to an interface package to a subsystem.

Example The body view corresponding to the reduced-visibility-graph of the simple system given in the Body View of Figure 7 is given in Figure 6. This has more columns than the corresponding Spec View shown in the same figure. The amount of abstraction during the design process can affect the shape of the View. However different methodologies would also be expected to result in different shapes. The only group {Error, Message, TextIO} can be easily picked out as it is the largest icon. It is the only icon in the third column.

A subgraph that is reduced to a single vertex in the above reduction is a maximal connected subgraph which contains a cycle on every vertex. These subgraphs, or *clusters* are of particular interest because of the effect of cyclic dependencies on maintenance.

Definition Let E be a set of edges. Then for each v_i for which there existsv_j such that $cyclic(E, v_i, v_j)$ holds

$$cluster(E, v_i) = \{v_j \mid cyclic(E, v_i, v_j)\}$$

Note that if $cluster(E, v_i)$ is defined then $v_i \in cluster(E, v_i)$ and if $v_j \in cluster(E, v_i)$ then $cluster(E, v_j)$ is defined and $cluster(E, v_j) = cluster(E, v_i)$.

Example For the set of edges$E$ in the body-graph shown in Figure 5,$cluster(E, TextIO) = \{TextIO, Error, Message\}$.

3.4 Displaying Connectivity

The spec and body concrete views defined above contain little information about the connectivity of the packages in the visibility graph of a system. The analyst needs access to more detail. We take one step towards doing this by looking at the *fan-in* and *fan-out* of vertices in the spec-graph

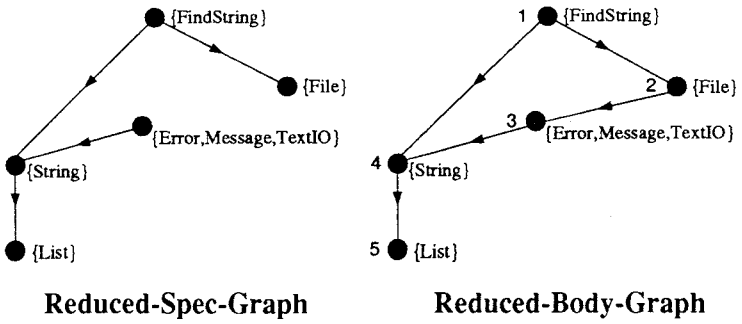


Figure 7 The Reduced-Spec-Graph and Reduced-Body-Graph for a Simple Software System.

and body-graph. We also want to be able to focus on subsystems of the software system, for example on the subsystem formed by a cluster. In this case, we will also be interested in the *external* connectivity of the subsystem components with the components of the rest of the system.

Definition Let G be a visibility-graph (V, E_s, E_b), V_c be a subset of V and v_i be a vertex inV_c. Then the following attributes are defined:

$$
\begin{aligned}
spec\_fan\_in(V_c, G, v_i) &= |\{v_j \mid (v_j, v_i) \in E_s\} \cap V_c| \\
spec\_fan\_out(V_c, G, v_i) &= |\{v_j \mid (v_i, v_j) \in E_s\} \cap V_c| \\
body\_fan\_in(V_c, G, v_i) &= |\{v_j \mid (v_j, v_i) \in E_b\} \cap V_c| \\
body\_fan\_out(V_c, G, v_i) &= |\{v_j \mid (v_i, v_j) \in E_b\} \cap V_c| \\
e\_spec\_fan\_in(V_c, G, v_i) &= |\{v_j \mid (v_j, v_i) \in E_s\} \setminus V_c| \\
e\_spec\_fan\_out(V_c, G, v_i) &= |\{v_j \mid (v_i, v_j) \in E_s\} \setminus V_c| \\
e\_body\_fan\_in(V_c, G, v_i) &= |\{v_j \mid (v_j, v_i) \in E_b\} \setminus V_c| \\
e\_body\_fan\_out(V_c, G, v_i) &= |\{v_j \mid (v_i, v_j) \in E_b\} \setminus V_c|
\end{aligned}
$$

where $\setminus$ represents set difference.

Note that if $V_c = V$, that is if we are considering the whole system, then the external attributes all equal zero.

We can extend our original definitions of the spec and body views of a system, using these attributes. In both cases there are no external vertices, and for the spec view we are only interested in the spec-graph.

Definition Let S be a software system with associated visibility-graph $G = (V, E_s, E_b)$ and reduced-visibility-graph $G^r = (V^r, E_s^r, E_b^r)$. We extend the previous definition of **Spec View** such that

- Each vertex $v_i \in V$, corresponding to a package, is represented by an icon displaying spec fan-in and fan-out calculated over the entire set of vertices V and for the graph G.
- The icons are arranged in columns so that the icon for vertex $v_i \in V$ is in the column given by $level(D, v_i)$, where $D = (V, E_s \cup E_b)$.

and **Body View** such that

- Each vertex $v_i \in V^r$, corresponding to a group of packages, P_i, is represented by an icon. The icon displays the number of packages in P_i together with spec and body fan-in and spec and body fan-out calculated over the entire set of vertices V^r and for the graph G^r.
- The icons are arranged in columns so the icon for vertex $v_i \in V^r$ is column given by $level(D^r, v_i)$, where $D^r = (V^r, E_s^r \cup E_b^r)$.

The icon we use is based on the body icon given in figure 8. The main feature required of the icon is that it depicts the quantities of the attributes clearly. The icon we used is a histogram with a base, where the height of each component indicates the number it represents. We considered a few other icons. One icon considered was a pie-chart, whose size was related to the total of the

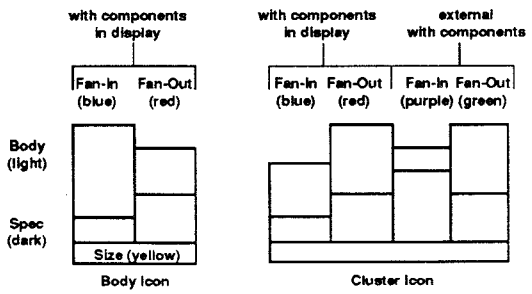
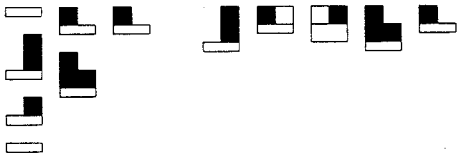


Figure 8 Icons Used to Represent Groups of Packages in the System Views.

values of the attributes shown. However, since we wanted to display the icons in columns we would have needed to limit the maximum size of the chart, so we discarded this idea. Another idea involved using colour intensity to indicate the magnitude of the attributes values. The problem with this idea is that we again have to limit the maximum values which can be distinguished, and there would be a problem for colour-blind people.

The additional information shown in these views enables us to pinpoint packages and clusters with high connectivity. However, we could display alternative metric values which were of interest. From the point of view of maintenance, modification to those with high fan-in values is most likely to cause changes to propagate, while those with high fan-outs require more packages to be examined in order to understand them. Another use of these views is that packages with high fan-in values, but low fan-out values, may have potential for reuse. A package which is going to be reused should be carefully checked for dependencies.

Example For the body-graph shown in Figure 5 we have the Spec and Body View shown in Figure 9 when we consider the relevant fan-in and fan-out values. In this simple system there are no packages which stand out because they have high fan-in or fan-out values.



Spec View **Body View**

Figure 9 A Detailed Spec-View and Body-View for a Simple Software System.

3.5 Cluster Views

As the clusters indicate areas of high-connectivity, we are interested in determining more about them. In the reduced-visibility-graph on which the body view is based, there is little information about the clusters and the properties of the packages in them. As a cluster is a subset of all the vertices in the graph, we find that all the attributes of the vertices in a cluster are of interest. As the cluster is formed by interconnected cycles in the body-graph, condensed to a single vertex in the reduced-body-graph, we cannot use our levelling function for clusters. Using this information we define another view to support the analysis of clusters.

3.5.1 Icons View

The first view we will define is the Icons View.

Definition Let S be a software system and $G = (V, E_s, E_b)$ be the associated visibility graph. Then the edges in the body-graph of the system are given by $E = E_s \cup E_b$. Let $v_i \in V$ be a vertex for which $cluster(E, v_i)$ is defined. Then we let $V_c = cluster(E, v_i)$ and define the **Icons View** of the cluster to be a concrete view with the following property:

- Each vertex $v_i \in V_c$ is represented by an icon displaying the values of the eight attributes described earlier for the graph G and the set of vertices V_c.

The icon used cannot be the same as that used in the spec and body views as it is required to display more information. As the information is an extension of that in displayed in the body icon, we use the Cluster Icon shown in Figure 8.

Because of the style of icon used, the main uses of this view are to determine packages within the cluster, which have high visibility or are highly visible, both to other packages within the cluster, and to packages external to the cluster. These packages can be analysed by focusing on them in other views or by using other tools [10, 3].

Example The icons view of a cluster with 12 elements is shown in Figure 10. The layout of the icons is arbitrary in this case, as levels have not been defined for the vertices in a cluster. There are two important things to note. The first is that all of the packages have an external fan-in value greater than zero, so the interface to the cluster is not clean. This means that more care would need to be taken when any part of the cluster is modified. The second is that while the connectivity within the cluster could be a lot higher, there are a few packages with high internal fan-in and fan-out values.

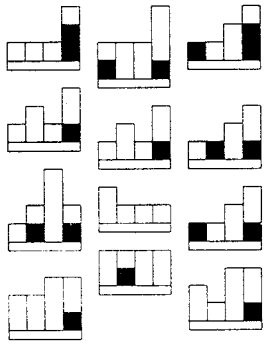


Figure 10 The Icons View of a Simple Cluster.

3.5.2 Graph View

For some purposes the Icons View of a cluster is sufficient. However if we wish to study the cluster to determine why it exists, and if it needs to exist, we need more information. The cluster may be of a small enough size that we can clearly view the edges between the packages, so we define another concrete view of a cluster.

Definition Let S be a software system and $G = (V, E_s, E_b)$ be the associated visibility graph. So the edges in the body-graph of the system are given by $E = E_s \cup E_b$. Let $v_i \in V$ be a vertex for which $cluster(E, v_i)$ is defined. Then we let $V_c = cluster(E, v_i)$ and define the **Graph View** of the cluster to be a concrete view with the following properties:

- Each vertex $v_i \in V_c$ is represented by an icon.
- There are two types of distinct, physical edges; spec-edges and body-edges, both of which may have an arrow at the end or at both start and end.
- There is an edge from $v_i \in V_c$ to $v_j \in V_c$ if there is an edge from v_i to v_j in the body graph.
- The edge used is a spec-edge if there is an edge from v_i to v_j in the spec-graph, and a body-edge otherwise.
- It is double-headed if there is an edge from v_j to v_i in the body-graph.
- The vertices have an initial arrangement where lines and vertices do not overlap, although they may cross. The analyst should be able to rearrange the vertices.

The icon used in the Graph View, is circular. The reason for this is that the icon can be small, and edges going to and from it can be clearly and easily drawn. They are originally displayed in a circular layout so that the lines do not overlap each other or any vertices. This view enables us to see which vertices are connected and rearrange them so that we may be able to determine how to improve the system structure. We place packages with external connections near the top of the graph so that they are visible. By also placing packages with low fan-in values near the top of the graph, we reduce the number of upward edges. We have included queries which highlight packages to assist the analyst with this task. Using queries we can highlight packages with low fan-ins, external connections or high internal connectivity and place them accordingly. The Graph View is most useful for clusters with low connectivity.

Example The Graph View of the cluster, whose Icon View is shown in Figure 10, is given in Figure 11. The circular picture shows how it was initially displayed and the other picture is the rearrangement performed by the analyst. The vertices have been given short labels, so that they can be identified. The rearrangement has been done so that the double-headed arrows can be clearly seen as they have been arranged to point across the page, and only a small number of arrows point up the page. In this arrangement, there is only one arrow pointing up the page. By removing this dependency and those which go both ways, the overall system design would become much simpler. We can apply further analysis to the packages using other tools to determine whether these modifications are practical.

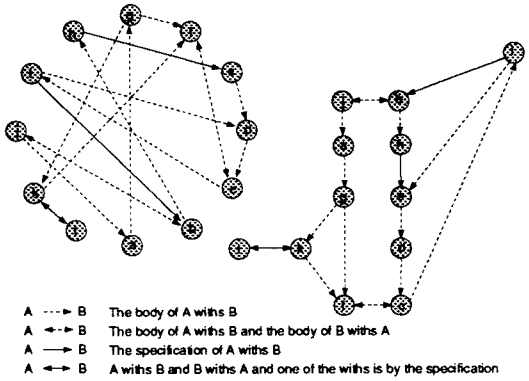


Figure 11 Two Forms of the Graph View of a Simple Cluster.

3.5.3 Table View

The Graph View is very useful, but as mentioned previously it is only useful for clusters with relatively low internal visibility. We have also developed a view based on the adjacency matrix for the cluster elements and are still exploring its potential.

Definition Let S be a software system and $G = (V, E_s, E_b)$ be the associated visibility graph. So the edges in the body-graph of the system are given by $E = E_s \cup E_b$. Let $v_i \in V$ be such that $cluster(E, v_i)$ is defined. Then we let$V_c = cluster(E, v_i)$ and define the **Table View** of the cluster to be a concrete view to be an adjacency matrix with the following properties:

- Each row of the matrix corresponds to a vertex$v_i \in V$. The corresponding column corresponds to the same vertex.
- There is an entry in a cell in the i^{th} row andj^{th} column if v_j is visible inv_i.
- There are two types of non-empty entries in the matrix, one for spec visibility and one for body visibility, when there is no spec visibility.

Example The table view of the cluster shown in Figure 11 is given in Figure 12. In this display the packages visible in the specification of Module_Nine is Module_Two, and the package visible in the body but not the specification of Module_Nine is Module_Four.

Once again, we do not expect an analyst to passively examine these views. As well as the query facilities described earlier, the analyst can edit the views, adding or deleting vertices or edges, to trial improvements in the static structure of the system. These improvements could eliminate clusters, make subsystem boundaries more clearly defined, make a component more amenable to reuse or simply reduce the overall connectivity of the system. While the analyst can trial these modifications to the abstract structure of the system, of course many other factors must be taken into account before the modifications can be accepted and applied to the system.

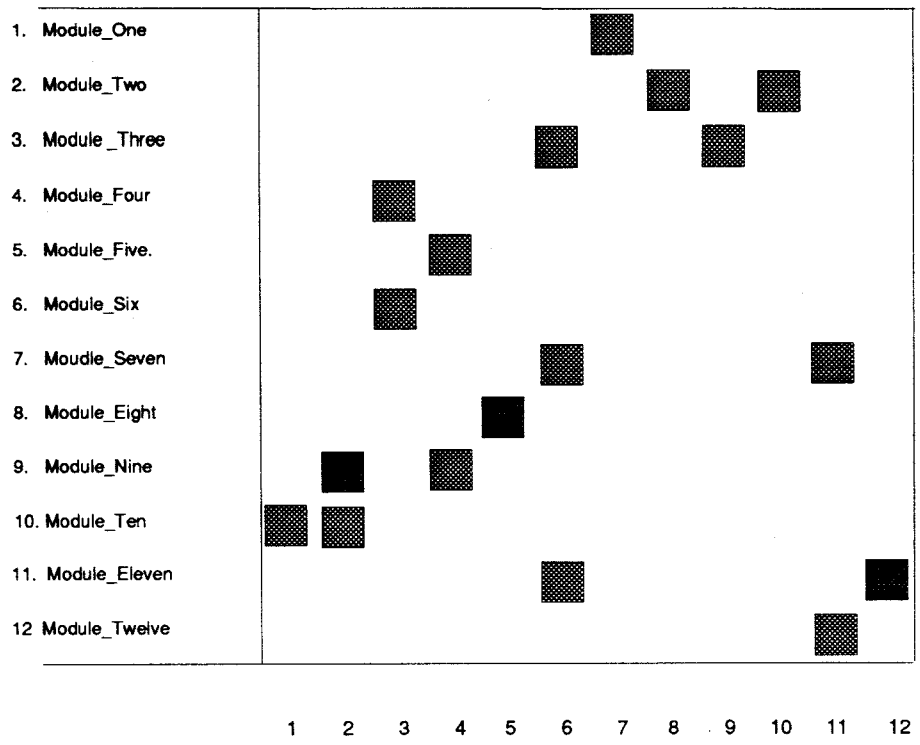


Figure 12 The Table View of a Simple Cluster

4 Observations

This section describes how we have used our ideas. It gives a description of a tool used to prototype our ideas and a tool which will make use of our views. It gives examples of how these tools have been used on actual systems and discusses areas we will investigate based on our observations using these tools.

4.1 Implementation

We initially implemented the environment described in the sections above in Quintus Prolog. We used the object-style graphics of ProWINDOWS, and used the Oracle RDBMS for long-term storage of the details of software systems being analysed. This development platform allowed us to rapidly develop a prototype environment in which we could experiment with different views.

The prototype environment has a number of features other than structural views which allow the analyst to actively explore the system being evaluated. The analyst can:

1. highlight components that **with** or are **with**ed by an indicated component.
2. select an icon and request that details of the corresponding component be displayed.
3. perform complex queries based on the properties of components. For example, all the components with a **spec_fan_out** greater than 5 and a **spec_fan_in** of 1 could be selected.
4. edit the views, adding or deleting vertices or edges, to trial improvements in the static structure of the system. These improvements could be aimed at
   a. eliminating clusters
   b. making subsystem boundaries more clearly defined
   c. making a component more amenable to reuse or
   d. reducing the overall connectivity of the system.

While the analyst can trial these modifications to the abstract structure of the system, of course many other factors must be taken into account before the modifications can be accepted and applied to the system.

Later we incorporated the views which were used consistently during analysis into another tool we have developed called SEE-Ada [10]. SEE-Ada is a Software Evaluation Environment for Ada which allows investigation of the structure of Ada programs. SEE-Ada provides views based on arbitrary units or subsystems, view extension via the **with**ing structure and easy access to source code. SEE-Ada currently contains views based on the **Body View** and the **Graph View**. The SEE-Ada icons depict the type of unit being represented and SEE-Ada's open framework allows metric values from external tools to be imported and overlaid on the system structure.

SEE-Ada uses an Ada parser to store information into an Oracle Database. Metric information from the external tools is also stored in the database. Information from this database is used by the prototype environment.

4.2 Analysis Experiences

We have used these tools in combination with others described in [10, 4] to analyse large (up to half a million lines of code) real-time embedded defence-related systems, a machine generated system and SEE-Ada itself.

Our reason for analysing systems is to improve the quality of code delivered under contract to the Australian Department of Defence. To achieve this we look at the software at various stages (*builds* or *baselines*) in its development and supply feedback to the developers.

When we receive a new build or baseline of a system we want to see how the system has changed. Of particular interest are (1) structural changes which can indicate instability in the design, (2) determining which of our recommendations have been followed up and (3) deterioration of general programming practices. We use the tool described to analyse structural changes and use SEE-Ada to overlay metrics on the software architecture as the first stage in determining deterioration of general programming practices. We use a combination of methods to view the effects of our recommendations. We found that by using these tools to assess the changes in the software, we also had some insight into the development process.

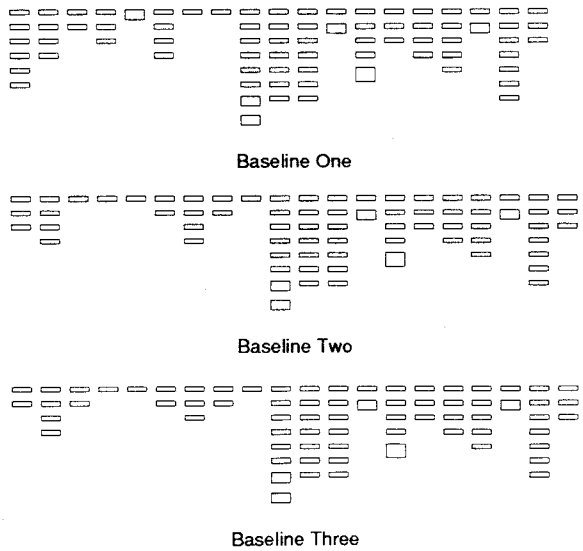


Figure 13 Several baselined versions of a Real System

Early attempts at assessing changes in software included comparing the names and sizes of files delivered for different versions. One system on which we used this approach appeared to have been quite radically changed with large numbers of added, deleted and modified files. However comparing the two builds of the software with our tool showed there was very little structural difference between the two versions. Further inspection showed that while a number of the files had been renamed, there was very little real structural change.

Example Figure 13 shows the difference between three sequential baselines of one of the systems analysed. In Baseline One we noticed that the code contained 6 clusters, and by looking at the modules in the first column, we determined that many test procedures and packages used for the automatic generation of documentation were delivered along with the source code. By looking at the connectivity information we were also able to determine that there were a large number of redundant **with**s. The existence of test procedures was discovered by finding out more details about the modules represented in the first column. One of the clusters had been eliminated — mainly by the removal of redundant **with**s — and most of the test and documentation packages were removed by Baseline Two, which was delivered 6 months later. By the third baseline, one more unit and some more redundant **with**s had been removed. The removal of redundant **with**s resulted in one package moving from level 7 to level 3. These changes all affected the first few levels. However, from about level 9, the design remained stable and the overall shape has remained similar.

Other areas in which this tool has helped us, and visitors who have used our tools, include the identification of modules with high or complex connectivity. In one system, with which maintainers were having difficulty, approximately forty modules where found to be in a single cluster. After the identification of the structure of this cluster the maintainers' view of the software was dramatically improved. We have found that our overall views of the system enable people to verify their mental image of the structure of the software and that they can often pick out the main packages from our display.

Our tool suite has helped us determine redundant **with**ing, clusters with only one element, redundant code, test code and modules which would not compile. For example, in one system we found an Ada specification which **with**ed itself. It can be seen from the few examples given here, that the tool has been of benefit to our analysis of large, actual, Ada systems.

4.3 Extensions of our Work

There are many ways in which this work could be extended. From our work on analysing actual systems, we have determined a number of views which are useful and have been included in a more robust tool and we have determined other areas in which our tool could be improved or extended.

Many of the ideas from this work have already been incorporated and extended in SEE-Ada. The Body View without attributes has been incorporated with the additional features of being able to zoom in and see the specifications, bodies and subunits individually named; to zoom out further so that the icons on the screen are merely dots and to see the entire system or only a specified group of subsystems. The Graph View, also with zooming, is available in SEE-Ada for any selected group of vertices, not only for clusters. These changes further improve the scalability of our views which, although readable in their prototype form, necessarily contain a large number of packages for large systems. The overlay of metrics is available with each of these views.

From the results of our analysis we have become convinced that characteristics of the process under which the software was developed need to be taken into account during analysis. For example, while analysing one system in the presence of a team manager, a few hotspots were highlighted. He immediately looked at who had written the code, which led to other modules being analysed. As well as classifying systems according to the process used during development and giving packages attributes such as who wrote them, we are interested in

- the classification of packages, involving classes such as abstract data type or definitions packages rather than those described in [11]

- classification mechanisms for clusters, with the aim of determining if any types of clusters are acceptable in delivered code and views based on groups of these classes
- and the effect of decoupling specifications, bodies and subunits.

Because of the number of ways our tools have been put to use, we are developing a tool called EASE to capture expert analysts' methods and enable them to be used by others. The approach we have taken is to use rules which take into account the characteristics of the software, and the environment in which it was developed, to determine which steps need to be taken for a particular analysis goal. Information about why a step should be completed and how to use different tools for the analysis are currently included.

There are several ways in which we have already started to extend the work described in this paper.

- We have considered other concrete views, some of which are based on quite different visualisation techniques such as the use of (simulated) 3–D.
- We are looking at higher level analysis views such as connections between subsystems, as well as looking at methods for determining subsystems. See [12, 13] for different approaches to determining subsystems.
- We are also working towards providing much more detailed views of static structure based on such things as the size of the name space of a package and the static procedure call graph.
- Finally we are looking at views designed for other analysis tasks such as performance based analysis. These could include views of dynamic call graphs and execution time usage. We also wish to consider views which would aid optimization and task analysis. (See [14] for some of the problems which occur when using tasks).

5 Conclusion

We have described an environment designed to support the analysis of large software systems. The environment presents the analyst with concrete views of a software system that can be examined, queried and edited. These concrete views are based on abstract views that capture the information required by the analyst to perform a specific analysis task.

We have highlighted some of the ways in which this work may be extended and given examples of how these views assist in the difficult task of analysing large, complex, Ada systems.

Acknowledgments
We would like to thank Gary Newsam, Chris Ekins and the members of the Software Engineering Group for their contributions to this work.

REFERENCES

[1] N. Fenton. *Software Metrics.* Chapman and Hall, 1991.

[2] Institute of Electrical and Electronics Engineers. IEEE Std 982.1-1988: Standard Dictionary of Measures to Produce Reliable Software, 1988.

[3] Dynamics Research Corporation. *AdaMAT/D Reference Manual*, 1990.

[4] R.J. Vernik and I. Turner. Techniques and Tools for Analysing Software Products. *Australian Computer Journal*, Aug 1992.

[5] Mark V Systems Limited, California, USA. *Adagen 1.6 User's Manual*, 1989.

[6] Hausi A. Muller and Karl Klashinsky. Rigi, A System for Programming-in-the-large. In *Abstracts of the International Conference on Software Engineering*, pages 19–20, 1992.

[7] Mariano Consens, Alberto Mendelzon and Arthur Ryman. Visualizing and Querying Software Structures. In *Proceedings of the International Conference on Software Engineering*, pages 138–156, 1992.

[8] American National Standard Institute. ANSI/IEEE Std 729-1983: Glossary of Software Engineering Terminology, 1983.

[9] R.J.A. Buhr. *System Design with Ada.* Prentice-Hall, Inc., New Jersey, USA, 1984.

[10] R.J. Vernik, I. Turner, C. Baker and S.F. Landherr. Automated Support for Assesment of Large Ada Software Systems. In *Proceedings of TRI-Ada'91*, page 237, New York, USA, 1991. The Association for Computing Machinery. This paper was distributed separately from the proceedings.

[11] G. Booch. *Software Components with Ada.* Benjamin/Cummings Publishing Company, Inc, 1987.

[12] Hausi A. Muller and James S. Uhl. Composing Subsystem Structure Using (K,2)-Partite Graphs. In *Proceedings of the Conference on Software Maintenance*, pages 12–19, 1990.

[13] A. Jaoua, J.M. Beaulieu, N. Belkhiter, A.C. Debaque, J. Desharnais, R. Lelouche, T. Moukam and M. Reguig. Rectangular Decompostion of Object-Oriented Software Architecture. In *Abstracts of the International Conference on Software Engineering*, pages 23–24, 1992.

[14] Karam G.M. and Buhr R.J.A. Starvation and Critical Race Analyzers for Ada. *IEEE Transactions on Software Engineering*, 15:829–843, Mar 1989.

Department of Defence

DOCUMENT CONTROL DATA SHEET

1. Page Classification	Unclassified
2. Privacy Marking/Caveat (of document)	

3a. AR Number	3b. Laboratory Number	3c. Type of Report	4. Task Number	
AR-009-029	ERL-0838-RN	Research Note	822545	
5. Document Date	6. Cost Code	7. Security Classification	8. No. of Pages	24
JULY 1994	254	* U U U	9. No. of Refs.	14

Document Title Abstract

S (Secret) C (Confi) R (Rest) U (Unclass)

* For UNCLASSIFIED docs with a secondary distribution LIMITATION, use (L) in document box.

10. Title

A VISUAL APPROACH TO THE ANALYSIS OF LARGE SOFTWARE SYSTEMS

11. Author(s)

Philip Dart, Gina Kingston,
Rudi Vernik and Stefan Landherr

12. Downgrading/Delimiting Instructions

N/A

13a. Corporate Author and Address

Electronics & Surveillance Research Laboratory
PO Box 1500, Salisbury SA 5108

13b. Task Sponsor

DSTO

14. Officer/Position responsible for

Security:.....

Downgrading:.....

Approval forRelease:...DERL.....

15. Secondary Release Statement of this Document

APPROVED FOR PUBLIC RELEASE

16a. Deliberate Announcement

No Limitation

16b. Casual Announcement (for citation in other documents)

[X] No Limitation [ ] Ref. by Author , Doc No. and date only.

17. DEFTEST Descriptors

Computer programs
Software evaluation

18. DISCAT Subject Codes

1205

19. Abstract

Analysing a large software system is a difficult and time consuming task, even with current automated assistance. Measures taken of software systems may be too abstract, applicable only to a particular class of system or simply too subjective. This paper examines problems associated with using existing tools and measures, and then presents an alternative approach to the analysis of large software systems based on system views that have been designed for a specific analysis task. We detail various abstract and concrete views of static system structure, and describe a software analysis environment developed to experiment with this approach. The concepts reported in this paper were developed during early enabling research undertaken as part of the SMAT task. Many of the concepts have already been included in other components of this task, including the SEE-Ada tool for software evaluation.