Finding Components in a Hierarchy of Modules: 
a Step towards Architectural Understanding

Jean-François Girard
Fraunhofer Institute for Experimental Software Engineering
Sauerwiesen 6
D-67661 Kaiserslautern, Germany
girard@iese.fhg.de

Rainer Koschke
University of Stuttgart
Breitwiesenstr. 20-22
D-70565 Stuttgart, Germany
koschke@informatik.uni-stuttgart.de

Abstract
This paper presents a method to view a system as a hierarchy of modules according to information hiding concepts and to identify architectural component candidates in this hierarchy. The result of the method eases the understanding of a system’s underlying software architecture. A prototype tool implementing this method was applied to three systems written in C (each over 30 Kloc). For one of these systems, an author of the system created an architectural description. The components generated by our method correspond to those of this architectural description in almost all cases. For the other two systems, most of the components resulting from the method correspond to meaningful system abstractions.

1. Introduction
It is well known that programmer efforts are mostly devoted to maintaining systems [Corb89, Somm92]. A large portion of that maintenance effort is spent in understanding the program and data [Yau80]. Within this context, helping maintainers to understand the legacy systems they have to maintain could greatly ease their job.

One important first step to assist the maintainer is to provide him/her with a global overview of the system. This overview should indicate the main components of the system, how they are related, and provide some constraints on these relations. This type of overview is often called the system architecture. There are still debates about the definition of a system architecture, but most agree that it should include at least components and connectors (how components communicate) [Garl93, Perr92]. Hopefully, an architectural description of a system can guide a maintainer’s attention to the parts of the system which need to be understood in more detail to perform the task at hand.

Most of the software architecture community focuses on defining and experimenting with formalisms to capture architecture while the system is specified and developed [Dean95, Shaw95, Luck95]. However, there is a large body of existing code which needs to be maintained and would also benefit from an architectural description. Thus, there is a need to recover architectural descriptions for existing systems.

This paper presents a method of building a view of a system composed of a hierarchy of modules and architectural component candidates in this hierarchy.

Paper Overview
The remainder of the article is organized as follows: Section 2 presents the related research. Section 3 introduces definitions and gives an overview of the proposed method. The following three sections present the individual techniques which support the method. Section 4 presents three techniques to identify potential components: one to identify recursive routine groups, one to identify abstract data types and one for abstract state encapsulations. Section 5 introduces dominance analysis and discusses how it provides the core information used to group routines into modules and subsystems. The technique to identify abstract state encapsulations and abstract data types can still leave some global variables not associated with any module; section 6 introduces a technique based on the usage frequency to distribute them into modules. The next section compares the results obtained by the approach with human analysis. Section 8 presents the conclusions and proposes further research.

2. Related Research
The experience acquired by the reverse engineering community can contribute to recover architectural descriptions for existing systems. Visualization tools for interactive reverse engineering, like Rigi [Müll94], can be used to
present an architecture and get the maintainer’s input. Modularization techniques like the ones used in this paper can produce static architectural views. Informal information available in identifiers and comments [Bigg93] can serve as a heuristic to choose between alternative component decompositions. Plan recognition [Will92] could be used to identify the implementation of algorithms typical of an architectural style [Perr92].

A few teams have already started to work on reverse engineering to the architectural level [Gall96, Tone96]. Harris et al. [Harr96] and Fiutem et al. [Fiut96], use clichés to recognize architectural features from source code. Yeh [Yeh95] has used techniques to identify abstract data types (ADT) or objects and considers them component candidates.

This paper presents a method to organize a system as a hierarchy of modules. These modules are considered architectural component candidates. The proposed method differs from other architectural recovery efforts [Harr96, Fiut96] in that it uses the dominance relation on the call graph to create a hierarchical view of components and that it assimilates recursive routine calls as well as modules and their local routines with their unique callers as component candidates. The proposed method also distributes global variables in modules according to a new frequency distribution technique.

Our method also has similarities with previous work. As in the previous approaches, we aim at recovering an “as-built” architecture (description of the actual software organization [Harr95]), since it is closer to the maintainer’s task than an idealized architecture. Like Yeh [Yeh95], we recognize abstract data types and abstract state encapsulations (Yeh calls them objects) as atomic component candidates. However, the techniques used to extract these components differ.

3. Terminology and Method

This section introduces the method and the prototype which implements it. The terminology used in the description of the method will be defined first.

3.1. Terminology

This paper uses the following definitions:

- **A module** is an encapsulation of user-defined data types, global variables and routines. An essential property of a module is that it has an **interface** which defines which of these types, variables, and routines are visible and accessible to parts outside the module, and a **body** which contains the implementation.
- **A subsystem** is a set of related modules or routines subordinate to a single routine.
- **A call graph** is a graph where nodes correspond to routines and edges to calls. It represents the calling structure within a system.
- **An abstract state encapsulation (ASE)** is a group of global variables together with the routines which access them.
- **An abstract data type (ADT)** [Somm92] is an abstraction of a type which encapsulates all the type’s valid operations and hides the details of the implementation of those operations by providing access to the types exclusively through a well-defined set of operations.
- **An atomic component** is the smallest component which is significant at the architectural level. In this paper, it is either an abstract state encapsulation, an abstract data type, a mixture of ADT and ASE (called **hybrid atomic component**) or a group of mutually recursive routines.

However, these atomic components are not always explicit in the source code, because programming languages like C do not provide sufficient support to express them and software development does not always exploit them. For this reason, reverse engineering techniques are required to identify ADTs and ASEs, even if their internal representation is public.

3.2. Method

Module abstraction allows a programmer to hide the implementation details. In a way, it offers him the possibility of guiding the future maintainer’s understanding, introducing details only when they become relevant. However, most conventional programming languages do not have support for expressing the subsystem or module interface. Therefore, the programmer often cannot express them in the code directly.

The proposed method identifies some of the information which can be hidden in an overview of a system. It proceeds according to the following steps:

1. Identify atomic components.
2. Identify subsystems and support routines.
3. Distribute variables outside atomic components into subsystems.

A maintainer should validate the results of each step before proceeding to the next. A maintainer should interpret the results of the method.

The results of the proposed method support information hiding in the following manner: Each type of atomic component constitutes a chunk of information which is easier to understand as a unit. Similarly, a subsystem can
be seen as a black box offering a number of services to its users. The functionality of a support routine is subsumed by the functionality offered by its callers.

Prototype

The proposed method was implemented in Refine™ on top of the Refine/C™ environment [Rea95]. The prototype uses Rigi [Müll94] to present results to the user.

4. Detection of Atomic Components

This section presents the techniques used to implement the first step of the proposed method, the extraction of atomic components.

4.1. Abstract State Encapsulations

A simple heuristic to identify ASE candidates is to group global variables and the routines which access them. However, this leads to artificially-huge ASEs in case of global variables that are used frequently within the system. An example of such a case would be a global variable “errno” that indicates an error and is used by many routines. To avoid this effect, one might exclude such frequently-used variables after a statistical analysis and validation by a programmer. Other approaches use certain metrics for cohesion and coupling [Canf93] to overcome the erroneous detection of huge ASEs because of frequently-used variables.

To cope with the problems of huge ASEs, the prototype groups only those routines that are declared in the same file as the global variables of the abstract state encapsulation. The assumption thereby is that a programmer groups together accessor routines and state variables in one file and allows, probably for efficiency reasons, direct access to the state in addition to the accessor routines.

4.2. Abstract Data Types

The prototype identifies ADT candidates with the classic heuristic [Canf94] which groups a user-defined data type and the routines that use this type in their parameter list or their return type (function signature). This heuristic can often lead to large ADTs, because a routine which has two user-defined data types in its signature will join the two groups of routines around them. Once again, the prototype uses the file heuristic to solve this problem.

4.3. Groups of Mutually-Recursive Routines

Groups of mutually-recursive routines should be presented as a unit, since they usually cannot be understood without one another. They can easily be identified in a call graph as strongly-connected components, which can be detected by Tarjan’s algorithm [Tarj72] in linear time with respect to the number of nodes and edges.

5. Identify Subsystems and Support Routines

The previous section introduced techniques which identify atomic component candidates. This section introduces the dominance analysis which produces a hierarchical view of a system. An interpretation of this hierarchy identifies the routines which support the candidates produced in the last section and the subsystems which could contain these candidates.

5.1. Dominance Analysis to Detect Subsystems

Dominance analysis produces a tree which captures succinctly whether a routine implements a service for one or for many callers.

Definitions

A node, N, is said to dominate another node, M, in a directed graph, G, if each path from the root of G to M contains N. If N is a dominator of M and every other dominator N' of M is also a dominator of N, then N is called an immediate or direct dominator of M. The dominance relationship can be represented as a dominance tree where a node’s parent is its immediate dominator.

Technique

Cimitile and Visaggio [Cimi95] propose to apply dominance analysis on call graphs to identify candidates for reusable modules. In their approach they collapsed cycles before dominance analysis gets applied. This approach is applied here to detect subsystems and to give a context to the atomic components. The main difference is that not only cycles are collapsed, but any atomic component. The proposed algorithm involves the following basic steps:

1. Each atomic component candidate is collapsed to a node.
2. Dominance analysis is applied on the collapsed call graph.
3. The dominance tree is interpreted as described in section 5.2.

These steps can be explained with the example of Figure 1. Part (a) shows the call graph before the algorithm is applied. There are two sets of mutually-recursive routines {4,8} and {6,10,11} that are collapsed in step (1). The result is shown in part (b). The result of applying dominance analysis to part (b) is presented in part (c) of Figure 1.

If A is the only caller of B, A necessarily dominates B. For example, 5 is the only caller of 9. This implies that 9
could be hidden in 5. For example, in Pascal we could declare such a routine as a local procedure. If A dominates B, but is not its only caller, B must be visible to all other components that A dominates, since there is at least one node dominated by A and not equal to B that calls B.

5.2. Interpretation of Results

By looking at the dominance tree the maintainer can decide which routines should go into the same module. The following guidelines support this process:

- If A dominates B, B can be a support routine. If so, it should be hidden in A, since no routine outside the dominance subtree rooted by A calls it.
- If P dominates A and B, A and B should belong to the same module as long as they are semantically related.
- All members of an atomic component should be in the same module. Members used outside the atomic component should be listed in the module interface, all others hidden in the module body.
- The subsystem of a node A is the subtree of the dominance tree rooted by A.

This recovered architectural view can be automatically compared to the original physical structure of the source code. Resulting divergences should be analyzed by the maintainer to suggest improvements or an alternative perspective on the system.

5.3. Dominance Analysis Applied to Real Systems

When applying dominance analysis on a suite of large systems written in C, several problems must be coped with. First, dominance analysis requires a unique root. In the case of C programs, this is usually the main routine. However, some systems consist of different executables and therefore have different main routines. The solution used by the prototype is to insert a unifying common root of all those main routines, called “the environment”. Second, some systems will have routines that are not connected to any main routine at all. In some cases, this could be dead code. However, in many systems this is due to routine pointers. For this reason, the prototype assumes a potential call of this code via routine pointers and therefore considers these routines as being called by the new environment node. A better solution would be to perform a dataflow analysis [Merl93] to collect all potential values routine pointers can assume and attach the corresponding routines to the call graph. This is part of our future work.

In case of application programming interfaces (e.g. X-window) or libraries, there is no root and it is reasonable, once again, to assume a call from the “environment” to each routine not otherwise called.

6. Remaining Global Variables Distribution

Some variables are not captured by the heuristic used to detect abstract state encapsulations. One way of putting them back in the system is to put them in a global “system state file”. However, if these variables are only used within a subsystem, they should be a part of the subsystem and not of the whole system.

The heuristics used here to select which subsystem S should contain a variable definition counts the number of references to variable V and selects the system with the highest number of references to V. If there are equal numbers of references, we choose one candidate arbitrarily. The number of references is computed with the following formula:

$$
\text{refs}(S, V) = \sum_{\text{dom}(S, P)} \text{refs}(P, V)
$$

where \text{dom}(S, P) means “S dominates P” and \text{refs}(S, V) is 1 if S references V and otherwise 0.

If there is just one subsystem using the variable, we attach the variable to the lowest possible level of the subsystem hierarchy. This enforces locality while ensuring that all parts of the subsystem that reference the variable have access to it.
Table 1: General characteristics of the three candidate systems of the case study.

<table>
<thead>
<tr>
<th>System Name</th>
<th>Version</th>
<th>Application</th>
<th>Lines of Code</th>
<th># of c files</th>
<th># user-defined types</th>
<th># global variables</th>
<th># user-defined routines</th>
</tr>
</thead>
<tbody>
<tr>
<td>aero</td>
<td>1.7</td>
<td>rigid body simulation</td>
<td>31 Kloc</td>
<td>36</td>
<td>57</td>
<td>480</td>
<td>488</td>
</tr>
<tr>
<td>bash</td>
<td>1.14.4</td>
<td>unix shell</td>
<td>38 Kloc</td>
<td>90</td>
<td>60</td>
<td>487</td>
<td>1002</td>
</tr>
<tr>
<td>cvs</td>
<td>1.8</td>
<td>concurrent version control</td>
<td>30 Kloc</td>
<td>51</td>
<td>41</td>
<td>386</td>
<td>575</td>
</tr>
</tbody>
</table>

If there is more than one subsystem using the variable, we attach the variable to the top of the selected subsystem and make the declaration of the variable externally visible.

7. Case Study

The analysis described above was applied to three medium-size C programs (see table 1 for their characteristics). Aero [Kell95] is an X window-based simulator for rigid body systems, bash is a Unix shell and CVS is a tool for controlling concurrent development. For this case study, the most interesting of these systems was Aero, since one of its developers produced a description of Aero’s architecture and compared the results of the proposed analysis to this description. For the other two systems, no descriptions of the general architecture were available. These systems were used more to gather quantitative information about components than to perform a qualitative comparison with a correct architectural description.

The developer described Aero as composed of three major subsystems: an X Window-based editor to construct the model to simulate, the simulator itself, and an output system that visualizes the simulation in an animation window and saves data for a raytracer program. The developer subdivided the main components as shown in Figure 2. The components are only logically hierarchical; in Aero’s implementation they are highly intertwined.

![Diagram of Aero's components](image)

**Figure 2. Main components of Aero.**

7.1. Atomic Component Detection

In the case study, about half of the routines were grouped by atomic component detection (see Table 2) before dominance analysis was applied. Out of these atomic components, between 33% and 78% corresponded to ADT and ASE identified by software engineers who did not know the techniques proposed [Gira97].

<table>
<thead>
<tr>
<th>System</th>
<th># routines in atomic components</th>
<th>total number of routines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aero</td>
<td>277</td>
<td>488</td>
</tr>
<tr>
<td>Bash</td>
<td>501</td>
<td>1002</td>
</tr>
<tr>
<td>CVS</td>
<td>246</td>
<td>575</td>
</tr>
</tbody>
</table>

Table 2: Atomic components detections.

<table>
<thead>
<tr>
<th>System</th>
<th># ADT</th>
<th># ASE</th>
<th># hybrid atomic components</th>
<th>mutually recursive routines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aero</td>
<td>0</td>
<td>189</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Bash</td>
<td>59</td>
<td>143</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>CVS</td>
<td>33</td>
<td>107</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3: Results of atomic components detection.

The results of atomic component detection are listed in Table 3. For Aero, the analysis revealed almost only abstract state encapsulations. All but one atomic component were confirmed by the developer. Three related subsystems (RGB coloring, projection, and redraw) that are defined in one large file share the display and screen variables. Therefore, the analysis considered the three components as one, though the developer regards them as distinct.

In the two other systems, the atomic component detection also found useful abstractions. For example, CVS uses a large hashing module. The implementation of the hash table uses two static global variables as caches for lists and nodes. The atomic component detection was able to identify this local subsystem within the hash table implementation.

7.2. Dominance Analysis

In the systems studied, our interpretation of the dominance tree revealed that between 31% and 36% of the nodes (detected atomic components and routines outside atomic components) are local to a single parent node.
That is, they provide functionality only to this node and therefore, can be understood in the context of this node.

In the dominance tree produced by this method, the nodes which are interpreted to be local to their parents, are at a level deeper than 1.

An often-used feature of the X window system is to connect a widget with a routine (named call-back routine) which should be called when the widget gets activated. The connection between a widget and its call-back routine is established by certain X window routines. As a result, the call of the routine is not visible in the source code.

As discussed in section 5.3, our method connects to the dominance tree, all routines that are not transitively called by main, such as these call-back routines. These uncalled routines and the main routines are then children of a node (called new root) introduced to be the root of the complete system.

Aero often uses call-backs. This is the reason for so many routines in the dominance tree of Aero in Figure 3 being dominated by the new root node. The new root node can be thought of as denoting the X window system in this case. By searching for X window routines that establish call-back routines, one could identify those routines directly dominated by the new root that are probably called by the X window system. All others are either dead code or called via function pointers. Function pointers were used by the three systems of this case study; this stresses the urgency of developing a method to handle this case.

Because abstract state encapsulation detection is applied before dominance analysis, some call-back routines are not directly dominated by the new root. In some cases, call-back routines share variables (e.g., variables that denote widgets) with other routines and are therefore grouped together in an abstract state encapsulation before dominance analysis is applied.

In Figure 3, the developer of Aero has overlaid the major subsystems of the system (see Figure 2) onto the dominance tree. The dominance tree has clearly identified several distinct parts of the major subsystems. This is an important step, even if it fails to put the corresponding parts together as one major subsystem. The combination can be performed by the maintainer during his interpretation. The method also identifies two supporting subsystems as local to other systems. The component Write World is local to Editor and Raytracer Output is local to Animation. This identification of supporting subsystems is significant, because it allows the maintainer to ignore large parts of the system until he needs to focus on them.

Dominance analysis clearly yielded a more structured view of the RCS subsystem of CVS. CVS is built on top of the revision control system RCS. The dependencies on

Table 4: Local vs Global Nodes in Dominance Tree

<table>
<thead>
<tr>
<th>System</th>
<th>#nodes in dominance tree 1.level</th>
<th>#nodes in dominance tree &gt; 1. level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aero</td>
<td>290</td>
<td>168</td>
</tr>
<tr>
<td>Bash</td>
<td>461</td>
<td>248</td>
</tr>
<tr>
<td>CVS</td>
<td>324</td>
<td>147</td>
</tr>
</tbody>
</table>

Figure 3: Dominance Tree for Aero
RCS are isolated in one large file. Within this file, dominance analysis identified a subsystem that implements a table of lines which is only used by a certain routine that annotates files. Another local subcomponent of the annotate procedure was a block allocator.

The proposed method offers limited support in understanding utility routines. Each subsystem of Aero has some utility routines, i.e., routines that are used to implement primary user routines but are of general usage. Since most of them are used in many places within each subsystem reachable from different call-backs, they are mostly directly dominated by the new root. Dominance analysis cannot distinguish them from primary user routines. Another problem with utility routines is that those that are only used once are considered to be local to their caller.

8. Conclusions and Future Work

In this paper, a method to identify architectural component candidates in a hierarchy of modules has been proposed. It identifies atomic component candidates which are abstract data types (ADT), abstract state encapsulations (ASE), and groups of mutually-recursive routines. This method uses dominance analysis on the call graph to identify support routines, modules, and subsystems.

A prototype tool implementing this method was applied to three systems written in C (Aero, bash, CVS; each system contains more than 30 Kloc). For one of these systems, Aero, an author of the system created an architectural description.

The subsystems generated by applying this method to Aero correspond, in most cases, to the main components in the true architectural description of Aero. In the case of CVS and bash, most of the components produced correspond to meaningful system abstractions.

The most productive technique to recognize atomic components was the extraction of ASEs. The other techniques yielded useful complementary information, which provides a context to these components, like their support routines, or identified other interesting, although less frequent, atomic components candidates.

Future work

The case study showed that additional factors need to be taken into account to widen the applicability of this method. For example, in order to perform a complete dominance analysis, routines which are reachable by routine pointer need to be attached at the correct position in dominance tree. We propose to use dataflow analysis [Merl93] to collect all potential values a pointer variable can assume.

Additional work is required to interpret the call-back mechanism in the context of the dominance tree. Clearly, more systems should be investigated in order to discover unexpected, but important cases like the call-back mechanism.

When variables are frequently used in large files, the kind of detection of abstract state encapsulations described in this paper can yield quite large components. This phenomenon could be observed in our case studies. We will investigate the use of a variety of techniques to recover the internal structure of large components. One such option is to apply dominance analysis on large abstract state encapsulations.

We need to assess the accuracy of the proposed techniques to recover atomic components. To achieve this goal, we will compare the results of the prototype's simple heuristic with other existing heuristics like the ones proposed by Yeh et. al. [Yeh95] and Ogando et. al.[Ogan94], and compare these results to atomic components software engineers would identify.

Acknowledgments

We would like to thank Jean-Marc DeBaud and Erhard Plödereder for their support of this project and their insightful inputs to this research. We wish to thank Hartmut Keller for providing us with a description of the architecture of Aero and for taking the time to discuss with us the results of our analysis. We are also grateful for Erhard Plödereder’s, Minna Mäkäräinen’s and Georg Schied’s suggestions for improvements during their reviews of early versions of the paper.

9. References


[Corb89] T.A. Corbi. Program Understanding: Challenge


