# Analyzing `xfig` Using the Bauhaus Tool

Jörg Czeranski, Thomas Eisenbarth, Holger Kienle, Rainer Koschke, and Daniel Simon
University of Stuttgart, Germany
bauhaus@informatik.uni-stuttgart.de

## Abstract

*We used the Bauhaus tool, which helps program understanding and architecture recovery, to analyze the xfig program. xfig's architecture was recovered and all of the (hypothetical) maintenance tasks described in the Developer Handbook were performed.*

## 1. Bauhaus

The **Bauhaus project** researches reengineering techniques to help program understanding of legacy code. Part of the program understanding task is the discovery of the system's architecture, which consists of components, subsystems, connectors, and constraints. Information about the system is exclusively extracted from the source code (because this is the only reliable source of information) in a semi-automatic way that involves the Bauhaus user (presumably a software maintainer or auditor).

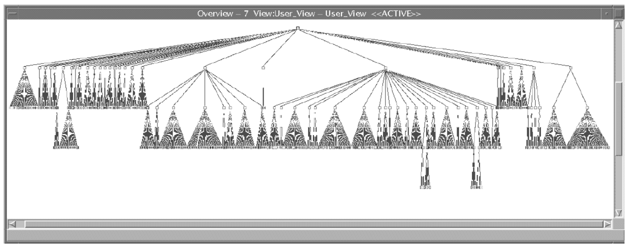

**Figure 1. View of recovered architecture visualized using Rigi**

Information extracted from the source code is represented in a resource graph (RG) [2], which abstracts from a particular source language by only representing global information such as call, type, and use relations. The RG information can be described by an entity relationship model. The entities are programming language constructs (e.g., functions, types, and variables) and abstract concepts, which typically have been extracted by certain analyses (e.g., abstract data types, components, and subsystems). Examples of relationships range from information that can be directly extracted from the source code (e.g., function calls) to more abstract concepts (e.g., communication between a client and a server). Entities are represented as nodes and relationships between the entities are represented as edges. The resulting graph can then be visualized and manipulated in Bauhaus with our extension of the graph editor Rigi (see figure 1).

Bauhaus supports a semi-automatic method to detect *atomic components* (ACs), also known as *logical modules* or *objects*. The analysis cycle consists of analysis application, metric ranking, presentation, and bookkeeping of detected components. The user controls the detection process by selecting analyses and metrics and by validating the candidates proposed by the automatic techniques. The task of the computer comprises the automatic analyses, computation of the metrics for the proposed candidates, presentation of the results, and bookkeeping of the user decisions.

An analysis selected by the user takes into consideration the components that were previously confirmed by the user (in the first iteration there are none). Thus, the analyses are applied incrementally. In each iteration, the user selects and combines different analyses to find components that could not be found by previous analyses. The process ends when the found components are sufficient for the task at hand or no further component can be found anymore.

We primarily used three analyses [2]: Same Module (SM) finds components where the types and globals are used by the subprograms and all parts are in the same file. Internal Access (IA) finds components where the internals of the types and globals are known to the procedures. Part Type (PT) finds components where the signature of the subprograms and the type derivation graph is used to cluster subprograms and types. SM and IA both find either abstract data types (ADTs) or abstract data objects (ADOs), whereas PT finds only ADTs.

Our **analysis team** consisted of four people. Due to time constraints, the team was divided in two sub-teams with two people each. One person in each sub-team was an experienced user of the tool; the other person was a novice. One sub-team analyzed the architecture of xfig, while the other sub-team tackled the maintenance tasks. There was no interaction between the two sub-teams.

## 2. Results

### 2.1. Architecture

xfig is written for the X Window System. Many global variables are used for communication (e.g., GUI callbacks). This is a questionable, but common design. We found that filenames mirror the role of the contents of a file and hence are useful as navigation hints. The file prefixes group the files into different categories (e.g., `e_`=edit, `d_`=draw, `w_`=widget). The files group edit operations (e.g., rotate, move, and scale)—which operate on graphical objects (e.g., arc, line, and compound)—into different files. This organization reflects a more functional than object-oriented style. As pointed out in [1], the functional style simplifies adding of new functionality to an existing data type, but makes it difficult to add new data types, whereas the object-oriented style has the opposite characteristic.

This means that adding a new operation is significantly simpler in xfig than adding a new graphic object (because the latter case has to touch all files dealing with operations). Since adding a new operation seems much more likely than adding a new object, the chosen design makes sense.

On the GUI side, handlers for one widget are grouped in one file along with the widget. The structure is fairly intuitive, reflecting the GUI components that are seen by the xfig user.

As we analyzed the system, we found that some of our analyses proposed rather big components due to communication via global variables. This revealed that complexity of communication is very high due to global variables—one has to understand the whole system to decide whether a change affecting the state of a global is harmless.

Because we had only one single version of xfig, we couldn't decide on the intactness of the initial design.

### 2.2. Maintenance Tasks

Reorganizing the tool bar involves changing the table `cmd_switches` in `w_cmdpanel.c`. New tables similar to `help_menu_items` have to be added. Splitting the commands "Load/Merge" from "Save As" and "Landscape" from "Portrait" requires writing new callback handlers—they should also be put into `w_cmdpanel.c`—or passing the choice as a parameter.

In order to add a new method for specifying arcs, the file `d_arc.c` needs to be modified. Functions similar to those already declared in this file need to be added to log and handle the three points set by the user. As a counterpart to `create_arcobject`, a new function needs to be added that takes the first point as center, the second one as starting point of the arc and the last point to compute the end point of the arc.

The bug that occurred when loading library objects is due to an endless recursion of a cycle in the call graph involving a call from `redisplay_region` to `move_object` via a function pointer.

## 3. Experience Report

### 3.1. RG Generation

All Bauhaus analyses that we used operate upon the RG representation. Our C front-end first generates an intermediate representation, which in turn is translated into an RG. On the first compile run, our front-end was able to process 92 out of 96 source files. After minor changes to the remaining four files we were able to compile all of them. We subsequently fixed our front-end (in about 3 hours) so that only three lines of code in one file had to be commented out to translate all sources. Total time until we got a first clean build of xfig (including the bug fixes) was about 5 hours. The last build (from scratch) took about 18 minutes.

### 3.2. Architecture Recovery

The RG that represents xfig consists of about 3200 nodes (among others there are 1526 subprogram, 1561 global variable, and 82 type nodes). The goal of the architecture analysis is to cluster as many nodes as possible together to form atomic components. However, without a time-consuming, in-depth analysis, a full coverage cannot be obtained. Due to the limited time, our goal was to obtain a fast, partial coverage by gathering the (partial) components that were directly proposed by the Bauhaus analyses. This partial architecture allowed us to make an intelligent guess on the full architecture.

As a first step towards architectural recovery, which turned out to be usually a good starting point, we ran three simple heuristic analyses (PT, SM, and IA). IA and PT both proposed a huge AC (among many other smaller ACs), which suggested that these analyses are not a good starting point for further refinement. Nonetheless they were useful, since IA revealed the heavy usage of global variables (the result was one big component) and PT revealed that user defined types do not play a major role in xfig (only small data type groupings were found in the beginning). SM produced best results making us confident that the file grouping is good. Since PT revealed a lack of types, we decided to run ADO detection analyses since they operate upon variables rather than types. We ran the two analyses that detect

ADOs—SM and PT—and intersected them to obtain more confident results. The intersection yielded useful ACs and covered about 20% of the nodes in the RG. After manually accessing the suggested ACs, we ran further analyses. The architecture recovery is an incremental process and after some components have been accepted, subsequent analyses produce better results because the nodes already accepted are not considered again. In the next analysis cycle, we ran PT, which grouped data types used for the graphical objects producing seemingly complete components and resulted in 31% coverage. In the last cycle, we ran SM and ended up with a 50% coverage. Due to time constraints we decided to stop at this point. (Further refinement could have been achieved by running more analyses, but at a certain point the coverage converges.)

Next, we organized the detected ACs hierarchically by grouping sets of ACs into superordinated ACs. There remained some polishing tasks uncompleted (e.g., some of the recovered components could have been further subdivided). The whole task took us about 5 hours.

### 3.3. Maintenance Tasks

The Bauhaus toolkit is currently specifically tailored for architecture recovery. Nevertheless, the statically extracted information and the cross-reference facilities of Bauhaus/Rigi are useful for other maintenance tasks, too. Whenever we needed additional information, we opportunistically used other available tools.

**Modifying the command panel** required identifying where the command panel is implemented. We searched with `grep` for the label of one of the command buttons. After locating the switch table, we used Bauhaus to find all functions that access it. These turned out to be only those functions that create and initialize X widgets, so we were reasonably certain that a change to the switch table won't have unexpected impact on the system. We used about an hour for this task.

**Adding a new method for specifying arcs** first required identifying where the existing way of specifying an arc is located in the code. To locate the feature, we searched with `grep` for the message that is emitted whenever the mouse cursor is on the arc drawing mode panel. This identified a table `mode_switches` for the mode panel. Using Bauhaus, we recognized all routines that access this table and browsed their source code. The analysis did not immediately reveal how the drawing routine is actually called. Our assumption was—inspired by the actual usage of xfig—that it must depend upon the current mode probably implemented as a global variable. Specifically looking for such a mode variable lead us to `cur_mode` accessed in the body of one of these functions. For `cur_mode`, we derived with the Bauhaus tool all other accessors, which eventually gave us the drawing routine for arcs. The rest of the maintenance task was done by reading the code of the detected file `d_draw.c` using the `emacs` tag facility to browse the source code. This task took us about one hour.

It was even our very first hypothesis that the **bug in loading library objects** is due to an endless recursion. The hypothesis was based on the fact that the program runs for a while and eventually crashes with a segmentation fault. When we increased the run-time stack, the program crashed later. In order to verify our hypothesis, we looked for cycles in the call graph, which can be automatically identified with Bauhaus. However, none of these cycles contained the suspected loading routine. Because the Bauhaus toolkit does currently not consider calls via function pointers, there could still be a recursive call including the loading routine. We verified this possibility by dynamic analysis (symbolic debugger) and could in fact find a cyclic call. Going back to the Bauhaus tool, we identified the cyclic call via a function pointer in the source code using the available information on sites where addresses of functions are taken. Altogether, we found the bug in about 20 minutes by a combination of dynamic analysis (symbolic debugger) and static analysis (call graph extraction and address-taken information).

## 4. Lessons Learned

The Bauhaus GUI, which is based on Rigi, has a few unpleasant properties. It is relatively slow, which can cause noticeable waiting periods for large graphs and hence sometimes disrupts the fluent use of the tool. Graphs with more than 500 nodes cannot be automatically layouted due to limitations of the external layouter `graphlet`. Rigi offers no drag-and-drop functionality, which forces the user to move nodes via the clipboard.

It proved useful for the maintenance tasks to use an opportunistic combination of tools (e.g., `dbx` and `emacs`) besides Bauhaus to solve the problem. A feature location facility (e.g., dynamic tracing of calls and subsequent visualization in Bauhaus) would have been useful for the maintenance tasks. Furthermore, the generated call graph in Bauhaus contains no calls via function pointers and, consequently, did not capture the cycle in the call graph caused by the xfig bug. A more sophisticated data flow analysis that takes points-to information into account would have probably been able to reveal the cycle.

## References


[1] T. A. Budd. Functional programming and the fragile base class problem. *SIGPLAN*, 33(12):66–71, December 1998.

[2] R. Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, University of Stuttgart, Germany, 2000. To appear.