# An Evaluation of Reverse Engineering Tool Capabilities

BERNDT BELLAY and HARALD GALL[*]

*Distributed Systems Group, Technical University of Vienna, A-1040 Vienna, Austria, Europe*

**SUMMARY**

**Reverse engineering tools support software engineers in the process of analyzing and understanding complex software systems during maintenance, reengineering, or rearchitecting. The functionality of such tools varies from editing and browsing capabilities to the generation of textual and graphical reports. There are several commercial reverse engineering tools on the market providing different capabilities and supporting specific source code languages. We evaluated four reverse engineering tools that analyze C source code: Refine/C, Imagix4D, SNiFF+, and Rigi. We investigated the capabilities of these tools by applying them to a real-world embedded software system as a case study. We identified benefits and shortcomings of these tools and assessed their applicability for embedded software systems, their usability, and their extensibility. © 1998 John Wiley & Sons, Ltd.**

KEY WORDS: reverse engineering; tool evaluation; reverse engineering tools

## 1. INTRODUCTION

Tool-support during maintenance, reengineering, or rearchitecting activities has become important to decrease the software engineers' time of manual source code analysis and to help focus on important program understanding issues. The size and complexity of commercial software systems requires automated reverse engineering support to facilitate the generation of textual and graphical reports (e.g. function reports, call trees) of the software system under study. But reverse engineering tools can only partly automate the program understanding process: Design recovery or architecture recovery activities still require the expertise of an engineer (Biggerstaff, 1989) and (Biggerstaff, Mitbander and Webster, 1994).

We evaluated four reverse engineering tools[1], each of which represents a different category of reverse engineering tools:

- Refine/C[2] is an extensible, interactive workbench for reverse engineering C programs. Refine/C is used to understand, evaluate, and redocument existing C code. It provides an API to use its

[*] Correspondence to: H. Gall, Distributed Systems Group, Technical University of Vienna, Argentinierstraße 8/184-1, A-1040 Vienna, Austria, Europe. E-mail: gall@infosys.tuwien.ac.at

[1] Refine/C Version 1.1a, Imagix 4D Release 2.7, Rigi V (Rigiparse V 5.4.3, Rigiedit V 5.4.3), SNiFF+ for C/C++ Version 2.3.1.

[2] Refine/C and Software Refinery are trademarks of Reasoning, Inc.

reverse engineering features to build customized analysis tools. Refine/C also provides programming access to its C parser and printer enabling to make extensions to the C domain model, grammar, or lexical analyzer. A customization of Refine/C through its API requires Software Refinery.

- Imagix 4D[3] is a comprehensive program understanding tool for C and C++ programs. It provides views to rapidly check and systematically study software at any level ranging from high-level design to the details of its build, class, and function dependencies. Imagix 4D presents this key information on software in a 3D-graphical format which enables the user to quickly focus on particular areas of interest.
- Rigi is a public domain tool developed in the Rigi Research Project at the University of Victoria (Hausi A. Müller). The main component is an editor called rigiedit. It is written in RCL, an extended version of Tcl/Tk, and supports viewing of parsed C, C++, PL/AS, COBOL, and LaTeX code (Wong, 1994). A C parser for generating the representation used in the Rigi system, called Rigi Standard Format (RSF), is also available. Rigiedit shows the correspondences of the entities that are generated by parsing the application and allows to edit these representations.
- SNiFF+ for C/C++ [4] is not a classical reverse engineering tool. It is an open, extensible and scaleable programming environment for C and C++ which also supports reverse engineering activities. SNiFF+ provides an efficient and portable environment with a comfortable user interface.

We investigated the capabilities of the above reverse engineering tools and identified their benefits and shortcomings in terms of applicability for embedded software, usability, and extensibility. The main focus was on the tool capabilities to generate graphical reports such as call trees, data and control flow graphs. For a more detailed presentation of the reverse engineering tools see (Bellay and Gall, 1996). This paper presents revised and enhanced results of (Bellay and Gall, 1997).

The experimental framework for evaluating software technology presented in (Brown *et al.*, 1996) states that a technology evaluation depends on the understanding of: 1) how the evaluated technology differs from other technologies; and 2) how these differences address the needs of specific usage contexts. In our case, the usage context was the recovery of architectural information from embedded software systems (Gall *et al.*, 1996, Eixelsberger *et al.*, 1997, 1998). For the evaluation we chose four significantly different representatives of reverse engineering tools to cover a wide spectrum of tools.

The paper is organized as follows: In Section 2 we give a brief overview of the case study. In Section 3 we define the tool assessment criteria and assess each tool in these terms. In Section 4 we present our results regarding the achievable software views, operating system dependencies, and shortcomings/benefits of reverse engineering tools. In Section 5 we draw some conclusions.

# 2. THE CASE STUDY

The evaluation was based on a part of a Train Control System that is a real-world embedded software system provided by an industrial partner. The system under study is one version of a family of systems which are safety-critical and have strong timing considerations. The software is

[3] Imagix 4D is a trademark of Imagix Corporation.
[4] SNiFF+ is a trademark of TakeFive Software GmbH.

programmed in two languages (C, Assembler) and has to run on different development and target environments. The system controls high speed train movement and realizes precision stops in metro stations.

The code size is approximately 150K LOC (Lines of Code), with the software implemented as state automatons as described in the SDL specifications of the system. Time critical parts are implemented in Assembler, while the rest of the software is implemented in C. The source code is well commented and the software documentation of the studied parts of the system were available for our evaluation.

# 3. TOOL ASSESSMENT

In this section, we define a set of evaluation criteria to be used in assessing the four reverse engineering tools. The definition of the criteria is based on our experience in applying these reverse engineering tools to the commercial case study.

## 3.1. Assessment Criteria for Reverse Engineering Tools

We introduce the following functional categories to assess the reverse engineering tools: analysis, representation, editing/browsing and general capabilities. In the following we describe each of these categories.

### *3.1.1. Analysis*

The parser is the core subsystem of every reverse engineering tool. The result of the parsed source code, i.e. the representation from which all the views are created, depends on the abilities of the parser. Parts of the source code that are not parsed or parsed incorrectly will affect all the generated views (Murphy, Notkin and Lan, 1996). The following is a list of criteria to be considered for a reverse engineering tool parser:

*Source types and project definition:*

1) Parsable source languages: Define which source codes can be parsed. The most common are C (ANSI C and K&R C), C++, COBOL, and FORTRAN.
2) Other importable sources: These represent other importable sources that can be used in the reverse engineering tool (e.g. test coverage-results which can be used for test verifications of the source code).
3) Project definition types: This represents how a project can be defined in the reverse engineering tool. Three methods of definition can be distinguished: file, directory, and makefile.
4) Ease of project definition: gives a measure of how easily a project can be defined.

*Parser functionality:*

1) Incremental parsing: Incremental parsing should be supported for systems that are changing during the reverse engineering phase (e.g. under development or maintenance). It provides the ability to parse only the parts that changed, thereby reducing the parse time.
2) Reparsing: Sometimes it is important to reparse the whole source code (e.g. when new macros are defined).

3) Fault tolerant parser: A fault tolerant parser provides the ability to parse incomplete, syntactically incorrect and uncompilable source code. This ability helps to parse source code fast and without changes, but has the disadvantage of not knowing if the parsed result represents the application generated by a compiler.
4) Define and undefine: The define and undefine preprocessor commands are important for parsing source code and should be supported by the parser. Two types of support can be distinguished: one is define and undefine only per project and the other one is additionally on a per file basis.
5) Preprocessor command configurable
6) Support for additional compiler switches: In addition to macros, compiler switches—such as include directories—should also be supported.

*Parsing functionality:*

1) Quality of parse error statements: The description of the parse error should help identify the source of the error and the reason why an error occurred.
2) Parse abortable, parse errors viewable during parsing, parse continued at error: These criteria can help reduce the time to parse the source code without errors to generate the representation in the reverse engineering tool.
3) Point and click movement from parse results to source code: This also helps to speed up the parsing process, but has to rely on the quality of the parse errors to enable the user to find errors.
4) Parsing results: The parsing results of the parser represent the correctness, completeness and type of information of the resulting representation of the source code in the reverse engineering tool.
5) Parse speed

### *3.1.2. Representation*

Representations are divided into textual and graphical reports, and properties of these reports, and are assessed based on their usability. For both kinds several general properties have to be considered.

*Textual (tabular, formatted, etc.):*

This provides a list with different textual reports, which are evaluated for each tool on their usability and completeness.

*Graphical (2-, 3-dimensional):*

This provides a list with different graphical reports, which are evaluated for each tool on their usability and completeness.

*General report properties:*

1) Speed of generation: The speed of generation of textual reports mostly depends on the amount of information to be shown. Graphical reports also depend on the layout algorithm used by the reverse engineering tool. Depending on the type and content of the reports, creation times up to a few minutes are very possible.
2) Filters, scopes, grouping: These three properties help to simplify the graph by showing only the part of interest. This is essential because graphical representations of applications are huge and therefore not useful in their complete representation. Filters are used to limit the view of a generated report (postprocessing) in contrast to scopes which limit the view of the report to be

generated (preprocessing). The grouping function is important to simplify the graph without losing the effect of the grouped entities on the rest of the view.

3) Point and click movement between reports allows easy navigation between reports based on a specific entity (e.g. function name, variable name).
4) Point and click movement from reports to source code allows to access the source code representation of a specific entity in a report. The browsing of the source code can be seen as switching the abstraction level and is done often during code comprehension.
5) Annotations help to remember certain aspects of the software directly where it is required without rebrowsing the part and also help to store the knowledge already acquired.
6) Static/dynamic views: This refers to the ability of reports to dynamically reflect the changes made to the source code inside the reverse engineering tool. An intelligent parser is required that is capable of parsing sources incrementally. In contrast, static reports have to be regenerated completely after reparsing the source code to reflect the changes.
7) External functions/variables: Functions and variables that are declared as external should be parsed and included in the reports.

*Textual report properties:*

1) Sorting: Textual reports may get huge and therefore should be sortable by user-specified criteria (e.g. name, file, or type).

*Graphical report properties:*

The following properties improve usability and readability of graphical reports.

1) Layout algorithms: Special layout algorithms can make a graph more readable; for example, minimizing the crossings of the arcs, showing the cohesion, or coupling of program elements.
2) View editable: Sometimes a view can be made more readable by manually editing it. This includes moving and deleting of entities that are currently not of interest (e.g. error routines). Grouping and filtering are further methods of editing a graph but are separate properties and therefore not included here.
3) Layered view: If a grouping of entities can be done recursively, a layered hierarchy is generated. A layered view of such a hierarchy can be shown either in one window (e.g. SHriMP view) or each layer of the hierarchy is displayed in a separate window.
4) A fisheye view helps to make a graph with a fixed size more readable by zooming in or out parts of the graph similar to a fisheye lens (Furnas, 1986).
5) SHriMP views (Simple Hierarchical Multi Perspective view) are a combination of layered views and the fisheye lens paradigm, enabling to show the layered structure in one view with the detail depending on the degree of interest.

### *3.1.3. Editing/browsing*

The editing/browsing capabilities are essential because the user often switches the abstraction level from the generated views to the actual source code. The text editor/browser should therefore provide means to facilitate browsing the source code.

1) Integrated text editor/browser addresses whether the reverse engineering tool provides an internal text editor/browser. Normally the integrated text editor/browser provides a better functionality with respect to the needs in a reverse engineering tool.

2) External editor/browser addresses whether external editors/browsers are supported by the tool.
3) Intelligent control of text editor/browser: For integrated editors an intelligent control is easier to provide but should also be supported for external editors. Intelligent control here means things, such as positioning at the appropriate line, opening a file for browsing or editing, etc.
4) Highlighting of the source code (e.g. currently active element)
5) Visualization functions facilitate the browsing of source code.
6) Speed of text editor
7) Search function: The text editor should provide a search function to find occurrences of patterns and easy methods to navigate among them.
8) The user interface should facilitate the browsing of the source code by providing short cuts or buttons for the functions most often used.
9) A history of the browsed locations should exist in the text editor. This is especially important if the text editor provides hypertext capabilities.
10) Hypertext capabilities: To browse the source code efficiently, the text editor should provide hypertext capabilities to jump to elements within or across source files.

### *3.1.4. General capabilities*

General capabilities span a wide range from supported platforms to online-help. We subdivide them into the following aspects:

1) Supported platforms
2) Multi-user support: Multi-user support for reverse engineering is not of such importance as in development tools because the application normally does not change and only one person may reverse engineer it. The need for multi-user support of course also depends on the size of the system under study.
3) Toolset extensibility: Most reverse engineering tools provide a fixed set of capabilities which cannot be extended. Although such a set might be quite large, it cannot foresee completely what different users may need and what new technologies may evolve. An open system often provides only a few composable operations and mechanisms for user-defined extensions. A closed system should provide a large set of built-in facilities and offers no possibility to extend the set. Reverse engineering tools can be extended either by extending the tool itself or by constructing integrated applications from a set of tools. Reverse engineering tools usually can be extended in the following directions: parser, user interface, and functionality. For constructing extensible integrated applications from a set of tools there are two basic approaches: tool integration and tool composition.
4) Storing capabilities: parsed source code, edited representation, tool state, selections, and views.
5) Output capabilities can roughly be grouped into printing, exporting, and automatic generation of documentation. The printing capabilities are important for the documentation process to show the results after applying the reverse engineering tool if the tool is not available. Furthermore it is important that the printing capabilities are intelligent (e.g. banner printing, scaling of graphics, usage of different line styles when no color is available, etc.) to produce useful results. The export capabilities are useful to be able to generate other results with the created representation of the source code (e.g. to create an entity relationship diagram in a case tool). The documentation capability can be seen as the creation of other views. Some tools provide the documentation of the

parsed source code automatically and others semi-automatically (e.g. macros for the generation). The output format can provide hypertext functionality to ease the use of the documentation.

6) History of browsed locations: A history of browsed locations not only for the text editor but also for all the other elements of a reverse engineering tool (e.g. reports) helps to go back to interesting parts already shown.
7) The search facility of a reverse engineering tool is an essential part: It can facilitate the task of creating groups of items (selections) and finding parts that are of interest. Not only patterns should be supported by the search engine but also properties of the entities used in the reverse engineering tool (e.g. scope of a function, location, size and other metrics).
8) Online-help

## 3.2. Assessment of the four Reverse Engineering Tools

The assessment for all criteria is done using one of three methods:

1. An enumeration of possible types,
2. Yes or no to classify the availability of a functionality, where no further classification is needed, or
3. A simple four-level scale, where the scale indicates whether a tool provides a functionality

   ++ excellent,
   + good,
   o acceptable, or
   - not at all.

In addition to these, a " / " is used if a functionality cannot be assessed. Table 1 shows the results for the four reverse engineering tools.

**Table 1 Assessment of the four reverse engineering tools**

# 4. EVALUATION RESULTS

This section gives an overview of the achievable software views provided by the four evaluated reverse engineering tools and briefly compares the call graph view of these tools. Afterwards a comparison of the tools and their extensibility is presented. The section concludes with general shortcomings of reverse engineering tool views, problems with reverse engineering tools due to operating system dependencies of the system under study, and some more general benefits and shortcomings of reverse engineering tools.

## 4.1. Achievable Software Views

The views that can be generated with the different reverse engineering tools are shown in Table 1 under "Representation." What is not shown, is that—although the evaluated reverse engineering tools provide similar kind of views—the representations and the possible manipulations, and thus the finally achievable views, differ quite a lot across the evaluated tools.

The figures used for illustrating the tools are from the case study, but had to be anonymized due to confidentiality reasons.

The views in *Refine/C* are either tabular reports or 2-dimensional graphs. All views can be focused on the interesting part of an application by selecting the scope of a view. Scope selection can only be done manually by selecting all interesting parts of the application shown in one view and then creating the other view. These selections get lost after creating a new view and no possibility exists to save a set of selections. Additionally, it is only possible to select the entities for the scope of the view in exactly one report and it is not possible to use a thus limited report to generate a new view (this is because the default scope is always the whole application).

The 2-dimensional graphs in Refine/C can be manipulated by moving the nodes. Nothing can be deleted or grouped together. This frequently results in very broad graphs if the whole application is viewed (see Figure 1). The views then are very large and have to be zoomed to identify the part of interest resulting in a loss of context. In such cases, the boxes are too small to view the whole names of entities making it impossible to distinguish entities with identical prefixes.

The views in *Imagix 4D* provide a 3-dimensional graph (except the control flow graph) and a text window viewing additional information and structure (see Figure 2).

By using three dimensions the graph does not get that broad. Using the rotate and zoom functions the graph can be viewed from all sides and the entities can be identified. The disadvantage of this smaller graph is that highly connected graphs get complicated and unreadable. The graphs can either be viewed top-down or bottom-up and the views are layered according to this selection. All entities viewed in the graph are also shown in context to the files and directories in the text window.

The graph can be manipulated by hiding or isolating selections made and also by grouping of the selected entities to one entity. This grouping function is very useful because functional groups can be combined to one entity and can then be observed in interaction with the rest of the graph. Another advantage is that such groupings help to make the graph smaller and more understandable. In contrast to Rigi, Imagix 4D does not provide layered views based on these groupings and these groupings cannot be saved between sessions. Selections can be made in any of the views and then used to limit the scope in the next created view (e.g. selecting a C file in the file view and hiding it creates a call graph without the functions in this file).

The selections can be made in many different ways ranging from manual selection of entities, automatic use of the UNIX command grep, automatic selection of all called entities starting from a particular entity, to a find function with many options. The selections cannot be saved, only one selection can be saved temporarily during one session. Created views made by using this functionality of Imagix 4D can only be saved temporarily during one session but get lost between sessions. In addition to these views inside Imagix 4D, reports can be created as part of the automated documentation process, which are formatted documents or hypertext documents (HTML).

*Rigi* creates 2-dimensional graphs that can be arranged either top-down, bottom-up or with two special algorithms: Sugiyama and Spring-layout. The Sugiyama-layout algorithm has a layered, tree-like layout that tries to minimize the crossings. The Spring-layout algorithm models the arcs as springs so that highly connected nodes tend to pull each other together and more isolated nodes tend to push each other apart (see Figure 3).

The graph can be edited manually, using filters and by searching and then deleting or hiding entities from the view. Together with the special layout algorithms well readable views can be

generated. Rigi also supports layered views and SHriMP (see Figure 4) views. Rigi allows to save both the edited representation of the source and the created views.

*SNiFF+* can create 2-dimensional graphical views (see Figure ***5***) and textual views which can be browsed. The graphical views show all the entities that one entity refers-to or is referred-by. This view is limited in the sense that connections can only go in one direction from an entity, therefore, resulting in many representations of one item (e.g. function) if the item is referenced somewhere else. These items are then marked in a special way. Because of this limitation of the views, highly connected entities cannot be identified and the reading of the views can get complicated with large graphs.

The entities in the graph can be filtered but no real manipulation of them is possible. The views created cannot be saved but remain the same until a new view is created or the tool to generate the view is terminated.

The achievable software views, as we have shown, are quite different. But they do not only differ in the kind of views that can be generated but also views of the same type can differ quite a bit between the reverse engineering tools. The next subsection gives an example of the possible diversity of software views based on the call graph.

### *4.1.1. Call graph*

Two functional views which usually can be generated by a reverse engineering tool are the functions report as a textual view and the call graph for graphical representation. These views show the functional coherence of the system under study. The call graph shows the call relations among a set of functions. Each function is represented by an item that has outgoing arrows to the function it calls, and incoming arrows from the functions that call it.

The call graph in *Refine/C* is called “structure chart.” To show functions that are defined as external (functions that are either programmed in Assembler or are part of a standard library) the call graph is created with the option “show externals” activated. Thus the resulting call graph is very broad. This is due to many functions that are either not called directly from the C source code or from functions in the standard libraries.

To view the entities in a readable form the call graph has to be zoomed (see Figure 1). As can be seen in the figure the entities are labeled with the initial substring that fits in the graphical label and thus cannot be distinguished.

*Figure 1. Refine/C call graph zoomed*

The call graph in *Imagix 4D* is smaller because of the use of the third dimension. The disadvantage here is that functions that are highly interconnected hide some of the information behind them. To see the entities that are at the “back side” of the call graph it has to be rotated. In large call graphs it sometimes is not possible to identify the functions in the middle of the graph (especially the connections to other functions). In the text window (on the right side of the call graph, see Figure 2) the entities are viewed in relation to the files and directories were they are defined. Functions that are not in one of the files are defined externally.

To create a more readable graph the view can be manipulated. The graph shown in Figure 2 has been created by hiding functions that are either not connected to any other function or are part of error routines.

*Figure 2. Imagix 4D filtered call graph*

The call graph in *Rigi* is created by viewing the graphical representation without the variables. This can be done by filtering or deleting them. The external functions (i.e. Assembler code) are not shown in the call graph, resulting in a smaller graph with less information. To create the call graph with the Spring-layout algorithm the functions that are not connected to the graph had to be deleted (the algorithm can be applied only to connected graphs).

The thus created call graphs show a grouping around the error functions. Resulting from this centering around the error routines, the call graphs are not showing the interesting parts clearly. The following graph (see Figure 3) was created by filtering these error routines. In this view, the centering around the main function and the module structure of the source code can be identified easily.

Rigi also provides the capability to generate layered (Harel, 1988) and SHriMP views. The layered views allow the grouping of functions (or any other type) into one entity. The result is a hierarchy of entities with the leaf nodes being the actual functions. The SHriMP view —Simple Hierarchical Multi Perspective view (Storey and Müller, 1995)— is a combination of nested graphs and the fisheye view paradigm (Furnas, 1986). The fisheye view used by SHriMP has the following features:

- Highly interactive zooming
- When one node is enlarged, the other nodes smoothly decrease in size to make room for the selected node
- Different areas of the graph may be inspected without permanently altering the graph
- A user may zoom multiple focal points and focal areas in the graph

The SHriMP views show the complete nested graph allowing to zoom-in or out of composite nodes down to the level of the atomic nodes. To create a SHriMP view the entities that were not of interest had all to be hidden first then a layered view was created of the case study. Figure 4 shows this SHriMP view after some nodes have been zoomed at different levels.

*Figure 3. Rigi call graph, filtered (Spring-layout algorithm)*

*Figure 4. Rigi SHriMP view of the Train Control System*

*SNiFF+* has no view which is a call graph only, but the cross referencer can be reduced to a call graph view by filtering the functions from all the symbols (see Figure ***5***). As stated before, the graph is viewed by starting at one entity and viewing all the entities that are either referred-to or referred-by this entity (to a given depth). To generate the call graph the “main” function, i.e. the one that calls most of the other functions has to be identified. Starting from this function all the functions that are called by it are shown. The graph is small because the external functions cannot be shown. It is interesting that the fan-in of functions cannot be identified well and since many references to a function are shown multiple times some information gets lost.

*Figure 5. SNiFF+ call graph*

## 4.2. Comparison of the evaluated reverse engineering tools

In contrast to the common assumption that reverse engineering tools have basically the same capabilities this was not the case for the four evaluated reverse engineering tools, at least with the exception that all tools provided a call graph.

Further, no evaluated tool provided all the functionality of the others or at least a majority. In fact the tools provided very different sets of capabilities, which may be partly due to the selection of the tools to represent a wide spectrum. Hence, none of the tools can be declared as the best or most useful reverse engineering tool in general. Every tool provided at least one advantage over the other tools.

### *4.2.1. Refine/C*

Refine/C provides an excellent parser in terms of the results of the internal representation (augmented abstract syntax tree). Especially in addition with Software Refinery it provides a good basis to extend Refine/C for specific applications. This is one reason why it is often used in the reverse engineering community. The generation of the projects, although it has to be done either manually or by a simple tool provided by Reasoning Systems, is flexible (e.g. define and undefine per file, exclusion of files, etc.). One major problem of parsing programs in Refine/C is that after a project has been parsed it cannot be reparsed, which sometimes is needed if changes to the project definition file have to be made during a session.

The user interface of the Refine/C workbench is restrictive (e.g. allows only one representation of each view, representations can only be placed on specific screen parts, etc.). Especially missing are capabilities such as a search engine, or an integrated editor, to ease the task of the reverse engineer.

The views that can be generated are the most important ones and only Imagix 4D provides more views.

The major drawback of Refine/C is the missing support to save scopes created or at least generating views from already limited ones. Also the manipulation of the graphical views should be supported in a better way (only the movement of the entities and browsing through the view is supported).

### *4.2.2. Imagix 4D*

Imagix 4D, like Refine/C, provides an excellent parser and the generation of projects is well supported by the tool (e.g. project definition by file, directory, makefile, reparse and incremental parsing, etc.).

The tool has a good user interface and is easy to use. The context sensitive online help and the tutorials contributed to make the learning curve a lot steeper than in the other tools. It provides the most number of views of all the evaluated tools and it owns the best set of supporting capabilities (e.g. search engine, integrated editor with highlighting and great browsing capabilities, etc.).

The generated views were sometimes too big and complex to be of real use, but because of a lot of capabilities to manipulate them (e.g. filters, scopes, groups, etc.) they could be well tailored to the task or problem at hand. Additionally the views could be saved during a session to be used for further reference or to be the starting point for a new view (the views can be used to save selections). Especially the queries for the source code that are provided proved very useful (e.g. to comprehend the program faster).

Another capability that is only provided by Imagix 4D is the automatic generation of documentation from the source code: It allows to generate HTML documentation which can then be browsed. What is also only provided by Imagix 4D is the import of additional data sources (graph profile data - gprof and test coverage data - tcov) that can then be related to the views (e.g. to show the test coverage results in a call graph).

Two major capabilities that are missing are a way to extend the tool and to generate graphical views that are also useful in printed form.

### *4.2.3. Rigi*

The major drawback of Rigi is the provided parser which can only parse functions and struct data types. This limits the views that can be generated mainly to functional views (call graph). Another problem is that the tool —because it is a research prototype— is not too stable.

The major advantages of the tool are that it features new technologies (e.g. layered views, SHriMP view, layout algorithms, etc.). The tool provides supporting capabilities (e.g. filters, metrics, groups, etc.) and it is extensible in some way. The created views are of good quality, partly due to the possible manipulations (e.g. filters, moving of entities) and the two layout algorithms which show additional information (e.g. cohesion and coupling). Rigi is the only tool that allows to save the generated views and representations.

### *4.2.4. SNiFF+*

The fast and fault tolerant parser in SNiFF+ is great when parsing source code that is incomplete or syntactically incorrect. But if correctness of the source code is of interest the fault tolerant parser can be a disadvantage. Only missing files are reported and this information can be found in a file. Also the generation of the projects in SNiFF+ turns out as rather inflexible: All the files have to be in one directory.

The main achievable view is the graphical cross referencer. The structure of this view is suitable for printing, but not for comprehension (entities can be found more than once in the cross referencer, but are then at least marked). SNiFF+ also provides good printing capabilities (e.g. banner printing).

The integrated text editor is almost as good as the one from Imagix 4D. It supports syntax highlighting and point and click movement which makes browsing the source code very easy. SNiFF+ is the only tool that also supports browsing between all generated views which comes in handy sometimes.

## 4.3. Tool extensibility

Tool extensibility is an important feature of many types of tools. This is also the case for reverse engineering tools, in which additional functionality often needs to be integrated to meet the specific constraints of a reverse engineering activity. In our case study it was important to extract information from the tool database to be integrated into another tool or with other recovered system information based on the Assembler source code. The following gives a short overview of the extensibility of the evaluated tools**.**

*Refine/C* provides an API to use its reverse engineering features to build customized analysis tools, such as by (Kontogiannis *et al.*, 1995) and (Whitney, *et al.*, 1995). Further, it provides programming access to its C parser and printer, to enable extensions to the C domain model, grammar, or lexical analyzer. A customization of Refine/C through its API requires Software Refinery. Furthermore, for small extensions, the output of the reports is customizable and can be converted easily.

*Imagix 4D* provides a scripting language to generate queries based on the capabilities provided by Imagix 4D (the auto queries are based on this). To use the information generated by Imagix 4D in

other tools, the output generated by the documentation functionality may be used, but the other tools need to cope with truncated long identifiers and the special format of the output.

*Rigi* (rigiedit) is programmable through a scripting language (RCL, an extended version of Tcl/Tk) and provides a customizable user interface (Whitney *et al.*, 1995). The separate parser can be replaced with another one based on the Rigi Standard Format (RSF) triples, that are binary relations representing the database.

*SNiFF+* is not extensible for reverse engineering tasks. It provides an interface for control integration via an API or an executable command.

## 4.4. System dependencies

The difficulties in reverse engineering the case study mainly resulted from two aspects which to some extent are typical for embedded software systems:

First, embedded software systems are usually programmed in more than one programming language. In the case study both C and Assembler were used. The reverse engineering tools examined are not able to parse Assembler code, so only the C parts could be parsed and studied, which results in incomplete views of the application. Furthermore, the C code contained externally defined functions and variables which are defined and instantiated in the Assembler part. As a result, some reverse engineering tools do not include these variables and functions in the generated reports. The exclusion of these variables caused problems because they are used for the main data flow in our case study.

Secondly, the case study was created for different development and target environments and was reverse engineered in again another environment. This resulted in some difficulties when parsing the C source code: Platform dependent parts of the application had to be supplied (e.g. header files) and platform properties had to be taken into account (e.g. file naming conventions, interrupt usage). Furthermore, application-specific knowledge was required to parse the application in a useful way, because macro definitions and additional files were used to generate different versions of the software (e.g. debugging, different application properties, etc.).

## 4.5. Shortcomings of achievable software views

Independent of the reverse engineering tool that generates views of the system, tool-generated views have—as manually generated views—some shortcomings.

Especially the graphical views can only show a part of the system tracing a specific problem. This is due to the fact that the view of the complete application is not usable in most cases, mainly because of the size of the system under study. The complete view of the system is typically unreadable and clustered (if the complete view is shown on the screen) or cannot be comprehended well (if only a part of the view is shown). To make the automatically generated views more readable and to relate them to a specific problem, quite some effort is required. Additionally, application-specific knowledge (e.g. which parts of the application are currently of interest) and domain knowledge (e.g. in this application domain the error routines are not of high importance and can, therefore, be excluded from the view) are important.

Many real-world systems hinder the generation of the complete views of the system (e.g. multi-language development, client-server applications, etc.). These views have to be generated either manually or by completion of the partly tool-generated view.

Another shortcoming of generated software views is that almost no focus on essential elements is possible across the different views. Specific problems cannot be traced easily across the views, because of the part of application shown and the different kind of information represented in the different views. The reverse engineer has to combine the different types of information shown by the views to generate the intended view of the application manually (e.g. recovered architecture, recovered design, etc.).

A further shortcoming of the achievable software views is the missing support of different layout algorithms for the graphical views (such as specific layout algorithms for printing, showing cohesion and/or coupling).

## 4.6. Benefits/shortcomings of reverse engineering tools

Many shortcomings of reverse engineering tools are due to the size of the system under study. The result are representations that get confusingly large. Especially the graphical views do not only grow in their size, they also get less readable (clustered, incomplete labels, etc.). Graphical views often need an unacceptable amount of time to be generated because of the layout algorithms.

Reverse engineering tools can generate several views, but to get a comprehensive “picture” of the application, these views are not sufficient. The tool-generated views often have to be completed manually because of different problems of reverse engineering tools: the inability to parse all source languages; client-server code cannot be related to each other automatically, etc. Supplemental manually generated views have to be created (some views simply cannot be generated by reverse engineering tools yet and others are either not supported by the individual tool or insufficiently generated, e.g. state transition diagrams).

Manually generated views cannot be created for all the specific problems or parts of the application traced (in contrast to the automatically generated views where this is at least possible). This makes it more difficult to relate the information found in the automatically generated views to the manually generated views.

Though it is said that applying reverse engineering tools to an application is an easy task, it requires application-specific knowledge (e.g. macros used for generating different versions of the software under study) and domain knowledge (e.g. safety critical systems) for many tasks during reverse engineering (e.g. parsing the source code, generating useful software views, etc.) making it not as easy and simple as it may seem.

Although some reverse engineering tools provide the ability to parse more than one programming language, they are not capable of mixed language support. Therefore, they cannot create projects or views with different source code languages.

The views only represent static information that can be found automatically in the source code. Additional application knowledge, therefore, is not included. A tool-support to include external application- and domain knowledge would enrich the views significantly.

Support for component identification, redundancy detection, and dynamic system analysis activities would also be desirable capabilities of reverse engineering tools.

# 5. CONCLUSIONS

There is no single tool that could be declared as the best of our evaluation. The reverse engineering tools evaluated are all quite different with varying strengths and weaknesses. They all provide good reverse engineering capabilities in different usage contexts. Figure 6 shows a final classification of the evaluated reverse engineering tools based on usability (for our task) and extensibility.

*Figure 6. Classification of reverse engineering tools*

***Refine/C*** provides an excellent parser in terms of the results of the internal representation (augmented abstract syntax tree). Especially together with Software Refinery it provides a good basis to extend Refine/C for special applications, which is one reason why it is often used in the reverse engineering community. The views that can be generated are the ones of major interest, but support in manipulating the graphical views more easily as well as in searching or saving selections are missing.

***Imagix 4D*** provides a large set of capabilities with a number of achievable views and also the capability to create documentation from the source code. The queries that are provided help to comprehend a program faster. Possibilities to extend the tool and to generate graphical views that are also useful in printed form should be included.

***Rigi*** features a lot of new technology, such as layered views, SHriMP view, layout algorithms, etc. and can be extended. One shortcoming is that the parser can only parse functions and data of type *struct*. This limits the views that can be generated mainly to functional views (call graph).

***SNiFF+´***s main view is the graphical cross referencer, which—due to its structure—is well-suited for printing. SNiFF+ provides also good printing capabilities. The fast and flexible fault-tolerant parser allows to parse incomplete or even syntactically incorrect source code. A shortcoming of SNiFF+ is that only the cross referencer can be seen as a software view of the system.

For the purpose of customizing a tool for a specific application Refine/C or Rigi (if the parser for C code is extended) can be recommended. To speed up program comprehension and to generate documentation and several views, Imagix 4D provides optimal capabilities. SNiFF+ is well-suited for software systems that are not completely parsable, for creating printable views and for browsing the system under study.

As a more general result, our tool evaluation showed that the performance and capabilities of a reverse engineering tool are dependent on the case study and its application domain as well as the analysis purpose. The study discovered that embedded software systems are difficult to analyze with current tool technology. Further, it showed that especially the graphical views and layouts require significant improvement and that a multi-language support is required for real-world applications.

# Acknowledgements

We are grateful to Wolfgang Eixelsberger, Lasse Warholm and Haakon Beckman for their support in analyzing the case study.

This work was supported by the European Commission within the ESPRIT Framework IV project ARES (Architectural Reasoning for Embedded Systems), project no. 20477.

# References


Bellay, B. and Gall, H. (1996) *An Evaluation of Reverse Engineering Tools (TUV-1841-96-01)*, Distributed Systems Department, Technical University of Vienna, 39 pp.

Bellay, B. and Gall, H. (1997) 'A Comparison of four Reverse Engineering Tools', in *Proceedings of the 4th Working Conference on Reverse Engineering*, IEEE Computer Society Press, Los Alamitos CA, pp. 2-11.

Biggerstaff, T. J (1989) 'Design recovery for maintenance and reuse', *IEEE Computer*, **22**(7),36-49.

Biggerstaff, T. J., Mitbander, B. G. and Webster, D. E. (1994) 'Program understanding and the concept assignment problem', *Communications of the ACM*, **37**(5),72-83.

Brown, A. W. and Wallnau, K. C. (1996) 'A framework for evaluating software technology', *IEEE Software*, **13**(5),39-49.

Eixelsberger, W., Warholm, L., Klösch, R., and Gall, H. (1997) ´Software Architecture Recovery of Embedded Software`, in *Proceedings of the 19th International Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos CA, pp. 558-559.

Eixelsberger, W., Ogris, M., Gall, H., and Bellay, B. (1998) ´Software Architecture Recovery of a Program Family`, in *Proceedings of the 20th International Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos CA, pp.508-511.

Furnas, G. W. (1986) 'Generalized fisheye views', in *Proceedings of ACM CHI 86*, Association of Computing Machinery, NewYork N.Y., pp. 16-23.

Gall, H., Jazayeri, M., Klösch, R., Lugmayr, W. and Trausmuth, G. (1996) 'Architecture recovery in ARES', in *Joint Proceedings of the SIGSOFT '96 Workshops - Second International Software Architecture Workshop ISAW-2*, Association of Computing Machinery, NewYork N.Y., pp. 111-115.

Harel, D. (1988) 'On visual formalisms', *Communications of the ACM*, **31**(5),514-530.

Kontogiannis, K., Galler, M., DeMori, R., Bernstein, M. and Merlo, E. (1995) 'Pattern matching for design concept localization', in *Proceedings of the 2nd Working Conference on Reverse Engineering (WCRE '95)*, IEEE Computer Society Press, Los Alamitos CA, pp. 96-103.

Murphy, G. C., Notkin, D., Lan, E .S.-C. (1996) 'An empirical study on static call graph extractors', in *Proceedings of the 18th International Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos CA, pp. 90-99.

Reasoning Systems (1990) *Dialect User's Guide*.

Reasoning Systems (1990) *Refine User's Guide*.

Reasoning Systems (1991) *Intervista User's Guide*.

Reasoning Systems (1994) *Refine/C User's Guide*.

Reasoning Systems (1995) *Refine/C Programmer's Guide*.

TakeFive Software GmbH (1997) *SNiFF+ for Unix*.

TakeFive Software GmbH (1997) *SNiFF+ User's Guide*.

TakeFive Software GmbH (1997) *SNiFF+ Tutorial Guides*.

TakeFive Software GmbH (1997) *SNiFF+ Reference Guide*.

Storey, M.- A. D. and Müller, H. A. (1995) 'Manipulating and documenting software structures using SHriMP views', in *Proceedings of the International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, pp. 275-84.

Whitney, M., Kontogiannis, K., Johnson, J. H., Bernstein, M., Corrie, B., Merlo, E., McDaniel, J., DeMori, R., Müller, H., Mylopoulos, J., Stanley, M., Tilley, S. and Wong, K. (1995) 'Using an Integrated Toolset for Program Understanding', in *Proceedings of the 1995 IBM CAS Conference (CASCON ´95)*, IBM Canada Ltd., Toronto ON, pp. 262-274.

Wong, K. (1996) *Rigi User's Manual V 5.4.3*, Part of the Rigi distribution package, pp. 161.


### Authors' biographies:

**Berndt Bellay** is currently a research assistant in the Distributed Systems Group at the Technical University of Vienna, where he investigates architecture recovery issues within the ESPRIT project ARES (Architectural Reasoning for Embedded Systems) and does his Ph.D. He received his M.Sc. in computer science from the Technical University of Vienna in 1997. His research interests are software

engineering, reverse engineering, software architectures, and programming languages. E-mail: bellay@infosys.tuwien.ac.at

**Harald Gall** received the M.S. and Ph.D. degrees in computer science from the Technical University in Vienna, Austria. He is currently Assistant Professor in the Distributed Systems Group at the Technical University of Vienna. His research interests are directed toward software architecture, architecture recovery and reverse engineering, and software evolution. Dr. Gall is a member of the IEEE and the Austrian Computer Society (OeCG). E-mail: gall@infosys.tuwien.ac.at

# APPENDIX A: TOOL INFORMATION

***Refine/C and Software Refinery***

Reasoning Systems, Inc.
3260 Hillview Avenue Palo Alto, CA 94304, USA
Phone: (415) 494-6201; Fax: (415) 494-8053
E-mail: info-requests@reasoning.com
Internet: http://www.reasoning.systems

*Software Version*

Refine/C Version 1.1
Refine Version 4.0

*Manuals*

Refine User's Guide (Revised May 25, 1990)
Intervista User's Guide (Revised March 4, 1991)
Dialect User's Guide (Revised July 6, 1990)
Refine/C User's Guide (Revised May 2, 1994)
Refine/C Programmer's Guide (Revised June 13, 1995)

***Imagix 4D***

Imagix Corp.
3800 SW Cedar Hills Blvd., Ste 227 Beaverton, OR 97005-2035, USA
Phone: 1-503-644-4905
E-mail: info@imagix.com
Internet: http://www.imagix.com

*Software Version*

Imagix 4D Version 2.7

*Manuals*

Context-sensitive online help

***Rigi***

Hausi A. Müller, Department of Computer Science,
University of Victoria, P.O.Box 3055, Victoria BC, Canada
Phone: (604) 721-7630; Fax: (604) 721- 7292
E-mail: hausi@csr.uvi.ca
Internet: http://www.uvic.ca/rigi

*Software Version*

Rigiedit V 5.4.3
Rigiparse V 5.4.3

*Manuals*

Rigi User's Manual V 5.4.3 (Postscript)

Rigi User's Manual V 5.4.1 (HTML)
Rigi Command Language (RCL) V 5.4.3 (HTML)
Rigi C-Language Parser (HTML)

***SNiFF+***

TakeFive Software GmbH, 5020 Salzburg, Austria or
TakeFive Software, Inc., Cupertino, CA
E-mail: info@takefive.co.at or info@takefive.com
Internet: http://www.takefive.com

*Software Version*

SNiFF+ Version 2.3.1

*Manuals (Postscript and HTML)*

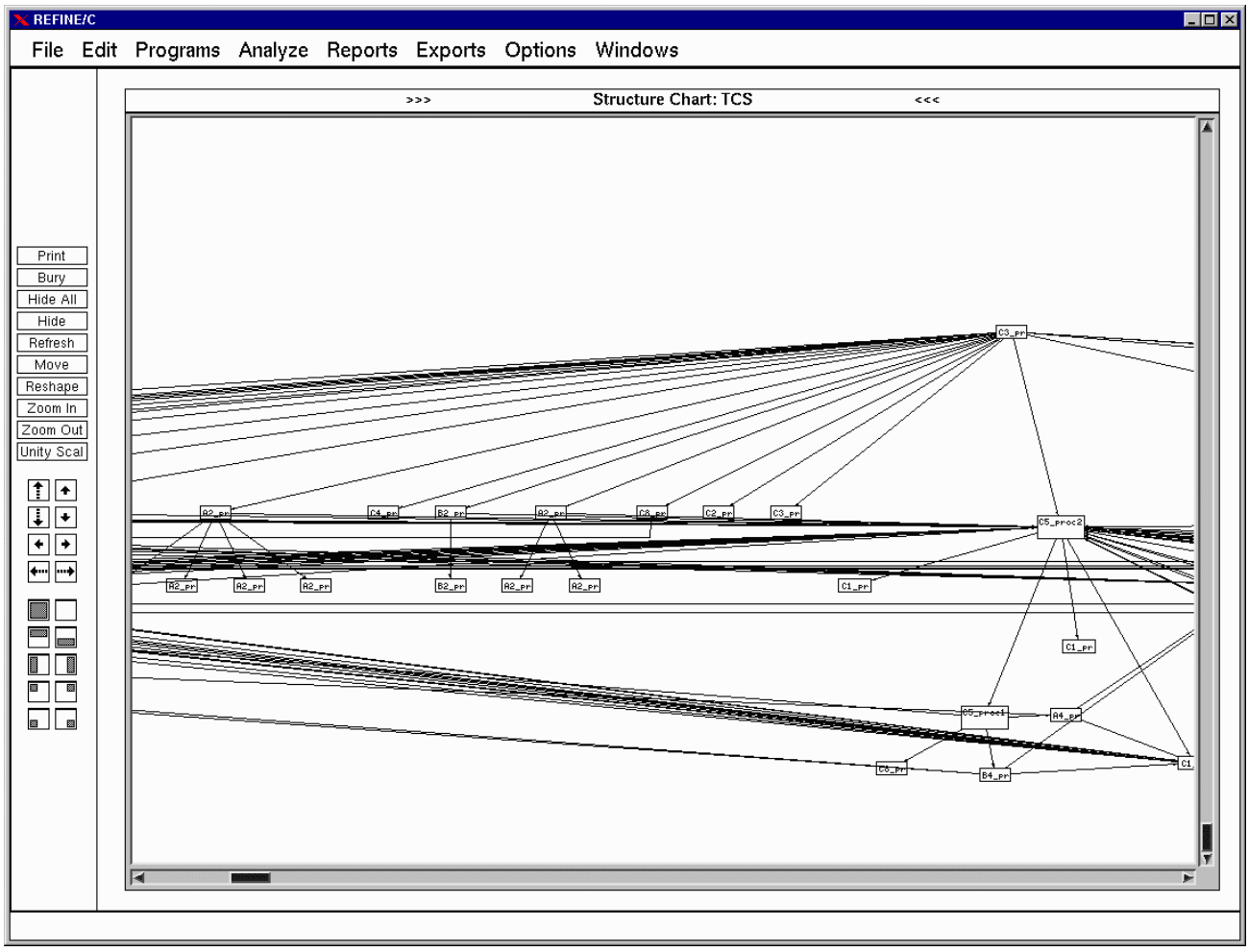
SNiFF+ for Unix
SNiFF+ User's Guide
SNiFF+ Tutorial Guides
SNiFF+ Reference Guide

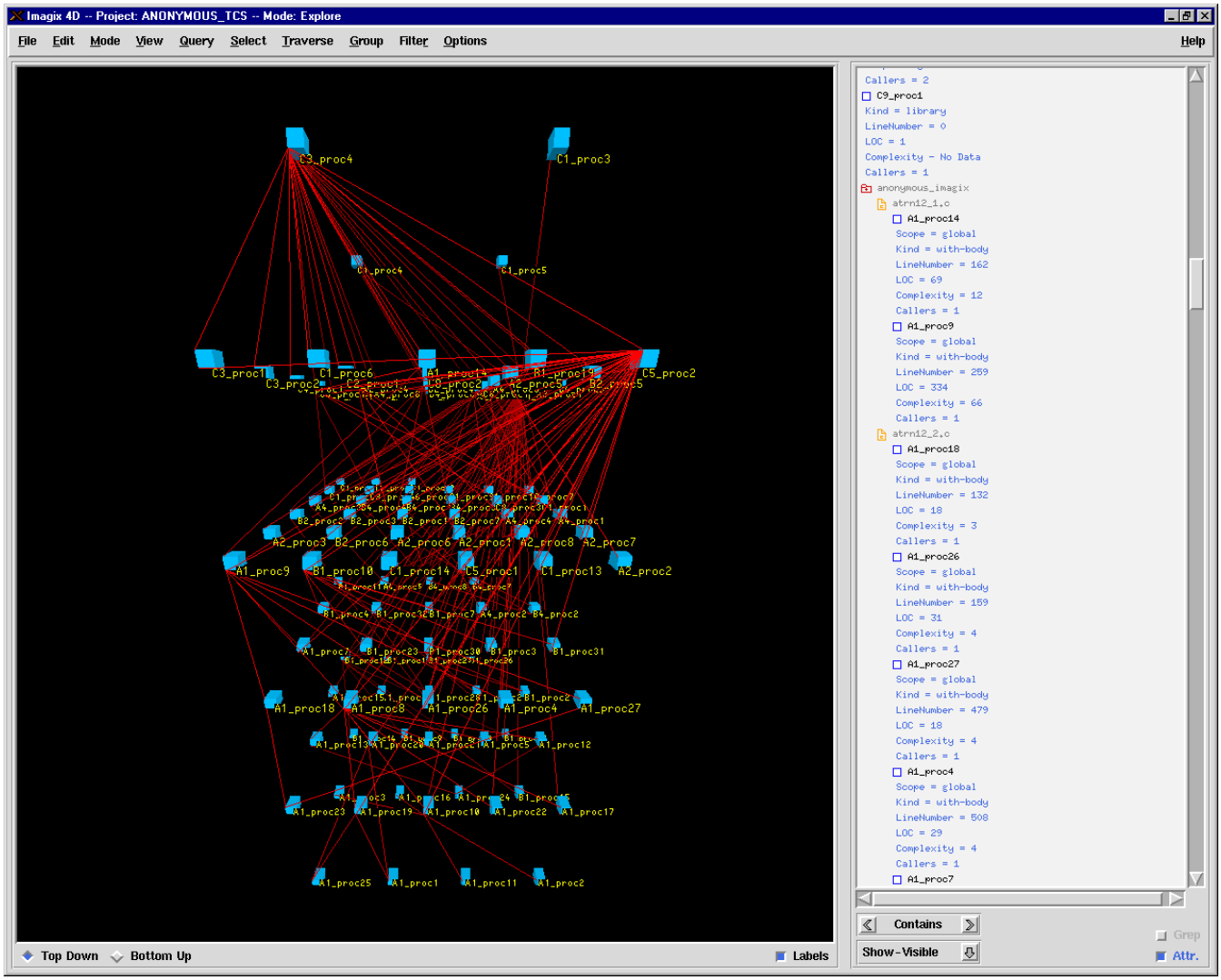
**Table 2. Assessment of the four reverse engineering tools**

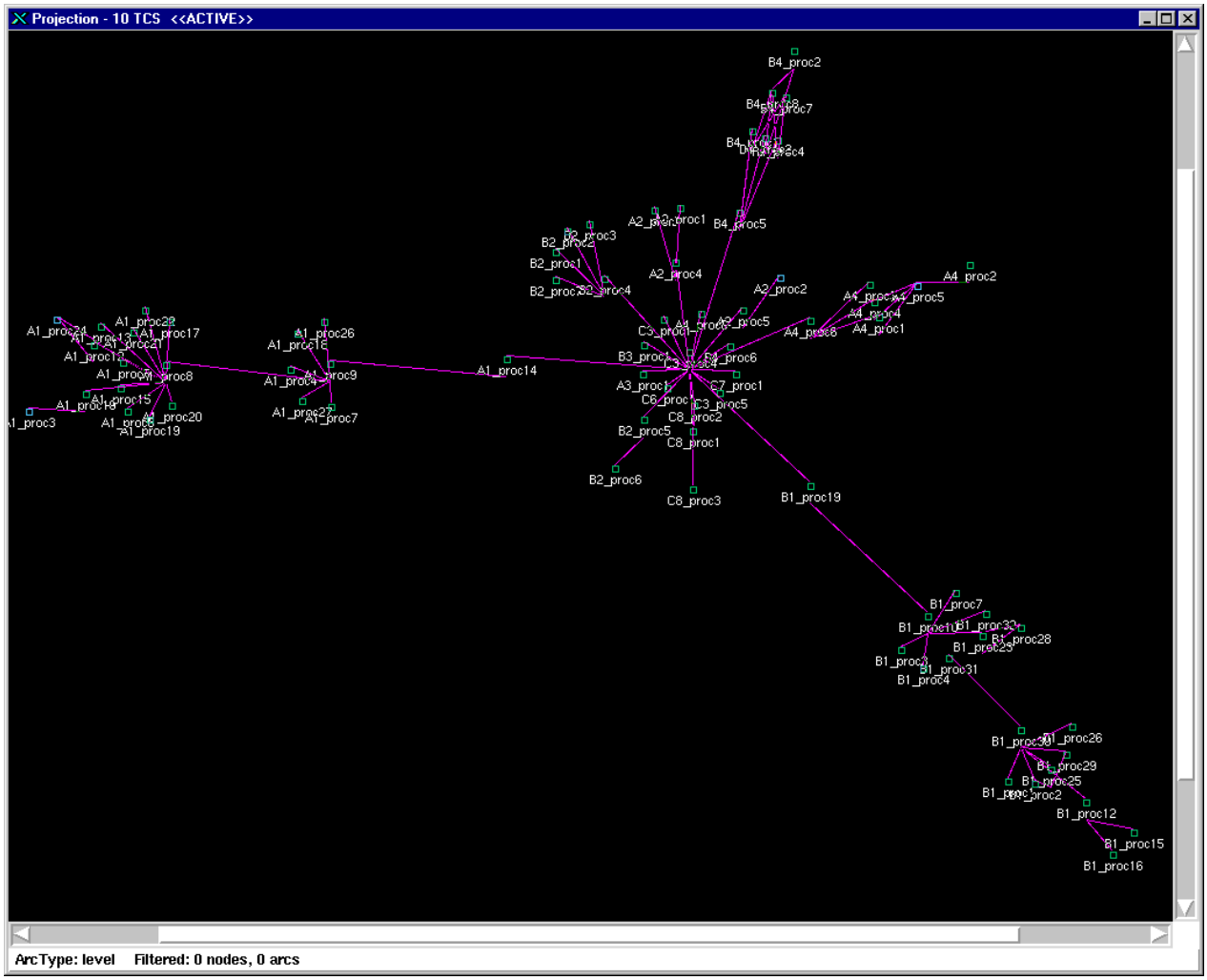
| Assessment Criteria | Refine/C | Imagix 4D | Rigi | SNiFF+ |
|---|---|---|---|---|
| General capabilities | o | o | o | o |
| Supported platforms | Sun Sparc; HP 9000/7xx; IBM RS/6000 | Sun Sparc; HP 9000/7xx; SGI | Sun Sparc; IBM RS/6000; Pentium PC | Sun Sparc; HP9000/7xx; HP9000/8x7; IBM RS/6000; IBM Power PC DEC Alpha; SGI; SNI RM; NCR; PC |
| Multi-user support | o | o | - | ++ |
| Toolset extensibility | | | | |
| tool integration | - | - | o | - |
| tool composition | - | - | o | o |
| parser | ++ | - | + | - |
| user interface | + | - | ++ | - |
| functionality | ++ | - | ++ | - |
| Storing capabilities | | | | |
| parsed source code storable | + | always | always | always |
| edited representation storable | - | - | + | o |
| tool state storable | - | - | - | + |
| selections storable | - | - | - | - |
| views storable | - | o | + | - |
| Output capabilities (printing) | | | | |
| storing of output | yes | yes | no | yes |
| output formats | ASCII, PS | PS, GIF | / | PS |
| intelligent printing capabilities | - | - | / | + |
| Output capabilities (exporting) | | | | |
| CASE tool | Software through Pictures | - | - | - |
| other reverse engineering tool | - | - | - | - |
| Output capabilities (documentation) | | | | |
| automatic generation | - | ++ | - | + |
| macros | - | - | - | + |
| output formats | / | ASCII, RTF, HTML | / | internal format |
| History of browsed locations | no | no | no | yes |
| Search facility | - | ++ | o | + |
| Online help | - | + | - | - |
| Analysis | ++ | ++ | o | o |
| Source types and project definition | | | | |
| Parsable source languages | C (ANSI), C (K&R) | C (ANSI), C++ | C (ANSI), C++, COBOL, PL/AS, Latex | C (ANSI), C++ |

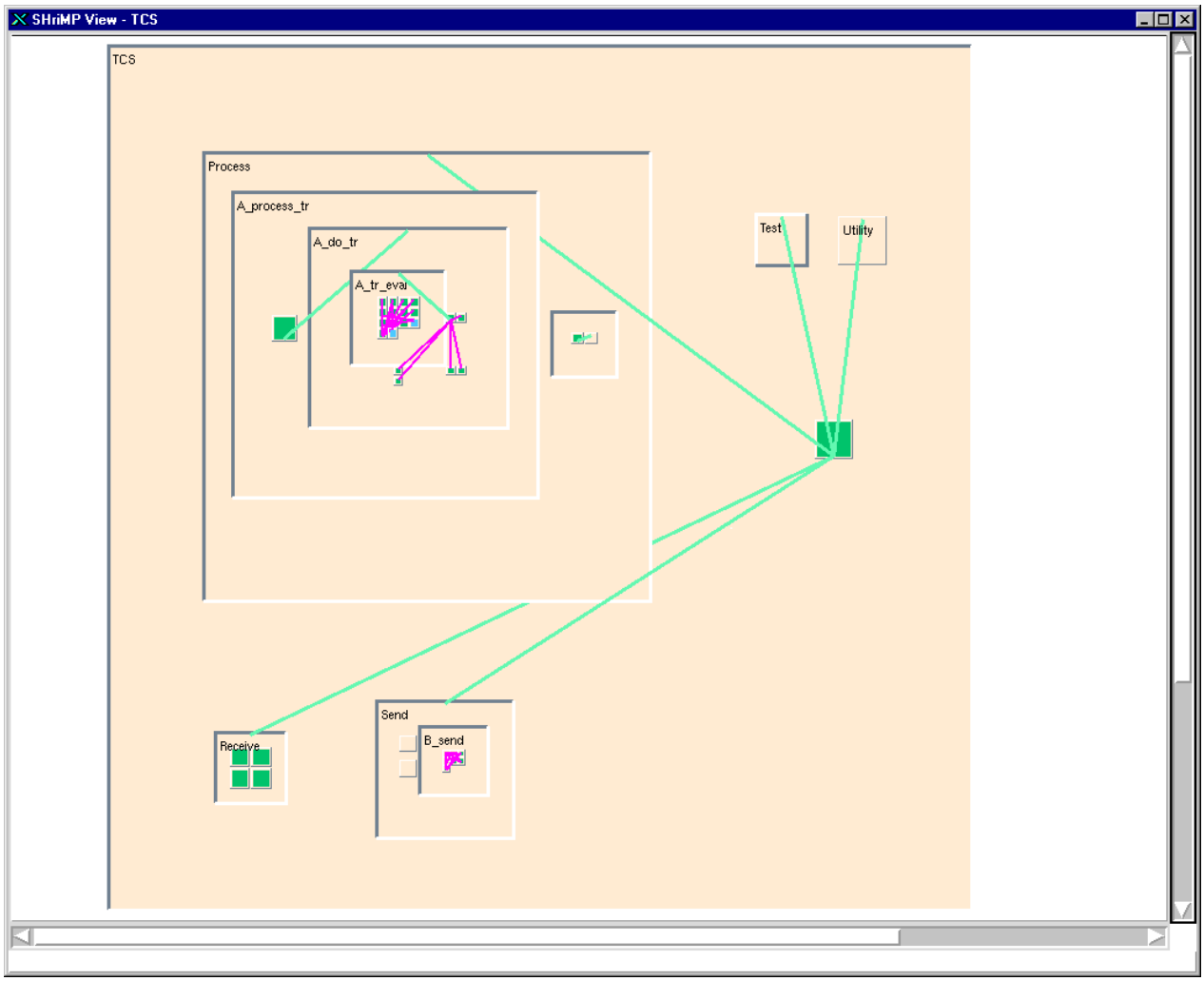
| | | | | |
|---|---|---|---|---|
| Other importable sources | / | gdb and tcov results | / | / |
| Project definition type | File | File, Directory, Makefile | File | Directory |
| Ease of project definition | o | ++ | - | + |
| Parser functionality | | | | |
| Incremental parsing | no | yes | no | yes |
| Reparse | no | yes | yes | yes |
| Fault tolerant parser | no | no | no | yes |
| Define and undefine | Project/File | Project/File | Project | Project |
| Preprocessor command configurable | yes | no | no | yes |
| Support for additional compiler switches | yes | yes | yes | yes |
| Parsing functionality | | | | |
| Quality of parse error statements | o | + | + | - |
| Parse abortable | yes | no | yes | no |
| Parse errors viewable during parsing | yes | no | yes | no |
| Parse continued at error | no | yes | yes | yes |
| Point and click movement from parse results to source code | yes | no | no | no |
| Parsing results | ++ | ++ | o | o |
| Parsing speed | o | + | + | + |
| Representation | + | ++ | + | o |
| Textual (tabular, formatted, etc.) | | | | |
| directory layout/organization | - | + | - | - |
| files | o | ++ | - | o |
| functions | + | + | - | o |
| types | + | + | - | o |
| variables | + | + | - | o |
| coding standard violation | + | - | - | - |
| pseudo code | - | - | - | - |
| Graphical (2-, 3-dimensional) | | | | |
| directory layout/organization | - | ++ | - | - |
| build map | - | ++ | - | o |
| call graph | o | + | + | o |
| control flow graph | - | ++ | - | - |
| data flow graph | + | + | - | o |
| data structure diagram | - | + | - | - |
| entity relation diagram | - | - | - | - |
| state transition graph | - | - | - | - |
| combined views | - | Calls and Variables, Full view | Full view | Full view |
| General report properties | | | | |
| speed of generation | o | + | + | ++ |
| filter | - | ++ | + | + |
| scopes | + | ++ | - | o |
| grouping | - | + | + | - |

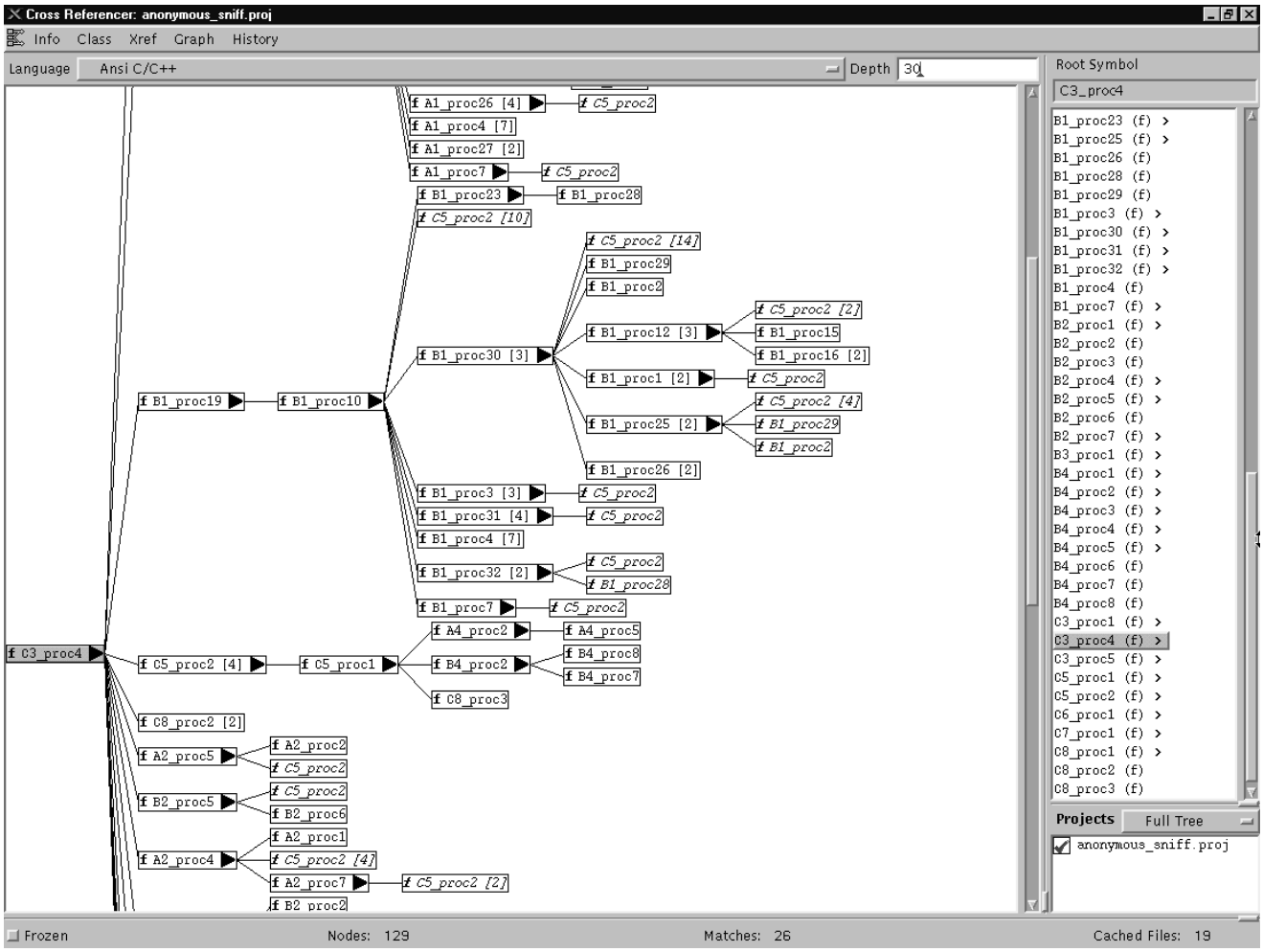
| | | | | |
|---|---|---|---|---|
| point and click movement between reports | - | - | - | + |
| point and click movement from report to source code | + | + | + | + |
| annotations | - | - | + | - |
| static/dynamic views | - | - | - | + |
| external functions/variables | o | ++ | - | o |
| Textual report properties: | | | | |
| sorting | + | - | - | - |
| Graphical report properties: | | | | |
| layout algorithms | - | - | ++ | - |
| view editable | o | - | + | - |
| layered view | - | - | ++ | - |
| fisheye view | - | - | - | - |
| SHriMP view | - | - | o | - |
| Editing/browsing | - | ++ | - | + |
| Integrated text editor/browser | - | ++ | - | + |
| External editor/browser | yes | yes | yes | yes |
| Intelligent control of text editor/browser | o | ++ | o | + |
| Integrated browser functionality | | | | |
| Highlighting | / | + | / | + |
| Visualization functions | / | ++ | / | + |
| Speed | / | o | / | + |
| Search function | / | + | / | + |
| User interface | / | ++ | / | + |
| History | / | + | / | + |
| Hypertext capabilities | / | + | / | + |

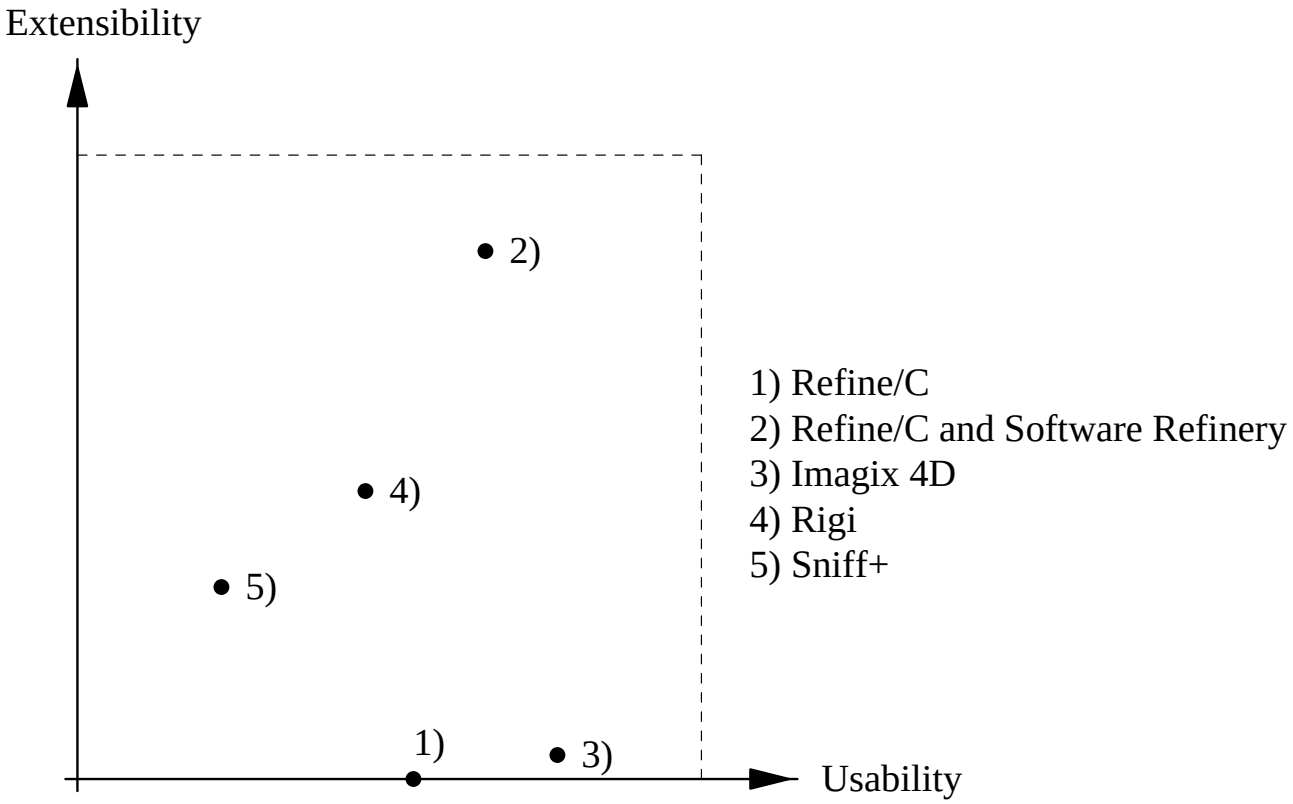

REFINE/C
File Edit Programs Analyze Reports Exports Options Windows
>>> Structure Chart: TCS <<<
Print
Bury
Hide All
Hide
Refresh
Move
Reshape
Zoom In
Zoom Out
Unity Scal
C3_PF
A2_PF
C4_PF
B2_PF
A2_PF
C8_PF
C2_PF
C3_PF
C5_proc2
R2_PF
R2_PF
R2_PF
B2_PF
A2_PF
A2_PF
C1_PF
C1_PF
C5_proc1
R4_PF
C6_PF
B4_PF
C1


Imagix 4D -- Project: ANONYMOUS_TCS -- Mode: Explore
File Edit Mode View Query Select Traverse Group Filter Options Help
C3_proc4
C1_proc3
C1_proc4
C1_proc5
C3_proc1
C1_proc6
C3_proc2
C5_proc2
B2_proc5
A2_proc3
B2_proc6
A2_proc6
A2_proc1
A2_proc8
A2_proc7
A1_proc9
B1_proc10
C1_proc14
C5_proc1
C1_proc13
A2_proc2
A1_proc18
A1_proc8
A1_proc26
A1_proc4
A1_proc27
A1_proc25
A1_proc1
A1_proc11
A1_proc2
Callers = 2
C9_proc1
Kind = library
LineNumber = 0
LOC = 1
Complexity - No Data
Callers = 1
anonymous_imagix
atrn12_1.c
A1_proc14
Scope = global
Kind = with-body
LineNumber = 162
LOC = 69
Complexity = 12
Callers = 1
A1_proc9
Scope = global
Kind = with-body
LineNumber = 259
LOC = 334
Complexity = 66
Callers = 1
atrn12_2.c
A1_proc18
Scope = global
Kind = with-body
LineNumber = 132
LOC = 18
Complexity = 3
Callers = 1
A1_proc26
Scope = global
Kind = with-body
LineNumber = 159
LOC = 31
Complexity = 4
Callers = 1
A1_proc27
Scope = global
Kind = with-body
LineNumber = 479
LOC = 18
Complexity = 4
Callers = 1
A1_proc4
Scope = global
Kind = with-body
LineNumber = 508
LOC = 29
Complexity = 4
Callers = 1
A1_proc7
Contains
Show-Visible
Grep
Attr.
Top Down
Bottom Up
Labels


Projection - 10 TCS <<ACTIVE>>
B4_proc2
B4_proc5
A2_proc4
A2_proc2
A4_proc2
A4_proc1
A1_proc14
A1_proc26
A1_proc9
A1_proc7
A1_proc20
A1_proc19
A1_proc17
A1_proc8
A1_proc15
B3_proc1
A3_proc1
C7_proc1
C3_proc5
C8_proc2
C8_proc1
C8_proc3
B2_proc5
B2_proc6
B1_proc19
B1_proc7
B1_proc28
B1_proc31
B1_proc4
B1_proc26
B1_proc29
B1_proc25
B1_proc12
B1_proc15
B1_proc16
ArcType: level
Filtered: 0 nodes, 0 arcs


SHriMP View - TCS
TCS
Process
A_process_tr
A_do_tr
A_tr_eval
Test
Utility
Send
B_send
Receive


Cross Referencer: anonymous_sniff.proj
Info
Class
Xref
Graph
History
Language
Ansi C/C++
Depth
30
Root Symbol
C3_proc4
B1_proc23 (f) >
B1_proc25 (f) >
B1_proc26 (f)
B1_proc28 (f)
B1_proc29 (f)
B1_proc3 (f) >
B1_proc30 (f) >
B1_proc31 (f) >
B1_proc32 (f) >
B1_proc4 (f)
B1_proc7 (f) >
B2_proc1 (f) >
B2_proc2 (f)
B2_proc3 (f)
B2_proc4 (f) >
B2_proc5 (f) >
B2_proc6 (f)
B2_proc7 (f) >
B3_proc1 (f) >
B4_proc1 (f) >
B4_proc2 (f) >
B4_proc3 (f) >
B4_proc4 (f) >
B4_proc5 (f) >
B4_proc6 (f)
B4_proc7 (f)
B4_proc8 (f)
C3_proc1 (f) >
C3_proc4 (f) >
C3_proc5 (f) >
C5_proc1 (f) >
C5_proc2 (f) >
C6_proc1 (f) >
C7_proc1 (f) >
C8_proc1 (f) >
C8_proc2 (f)
C8_proc3 (f)
Projects
Full Tree
anonymous_sniff.proj
Frozen
Nodes: 129
Matches: 26
Cached Files: 19


Extensibility
2)
4)
5)
1)
3)
Usability
1) Refine/C
2) Refine/C and Software Refinery
3) Imagix 4D
4) Rigi
5) Sniff+