Using an Integrated Toolset for Program Understanding

Michael Whitney Kostas Kontogiannis J. Howard Johnson
Morris Bernstein Brian Corrie Ettore Merlo James McDaniel
Renato De Mori Hausi Müller John Mylopoulos Martin Stanley
Scott Tilley Kenny Wong

Abstract

This paper demonstrates the use of an integrated toolset for program understanding. By leveraging the unique capabilities of individual tools, and exploiting their power in combination, the resultant toolset is able to facilitate specific reverse engineering tasks that would otherwise be difficult or impossible. This is illustrated by applying the integrated toolset to several typical reverse engineering scenarios, including code localization, data flow analysis, pattern matching, system clustering, and visualization, using a mid-size production program as the reference system.

1 Introduction

As the amount of legacy code currently in use increases, the importance of program understanding grows accordingly. *Program understanding* is the process of developing mental models of a software system's intended architecture, purpose, and behavior. There have been numerous research efforts to develop tools that provide assistance during the understanding process. These tools adopt a number of different approaches, including pattern-matching, visualization, and knowledge-based techniques. Despite successful results from each of these approaches, it is clear that no single tool or technique is sufficient by itself and that the software engineer can best be served through a collection of tools that complement each other in functionality.

Over the past three years, we have been developing a toolset, called RevEngE (*R*everse *E*ngineering *E*nvironment), based on an open architecture for integrating heterogeneous tools. The toolset is integrated through a common repository specifically designed to support program understanding [1]. Individual tools in the kit include Ariadne [2], ART [3], and Rigi [4]. Ariadne is a set of pattern matching and design recovery programs implemented using The Software Refinery from Reasoning Systems, Inc. ART (*A*nalysis of *R*edundancy in *T*ext) is a prototype textual redundancy analysis system. Rigi is a programmable environment for reverse engineering and program visualization. Tools communicate through a flexible object server and single global schema implemented using the Telos information model [5].

This paper shows how the reverse engineering capabilities provided by the RevEngE toolset are used to aid program understanding. This is done by applying the integrated toolset to selected reverse engineering analysis scenarios representative of actual program understanding tasks. No prior knowledge of the subject system's source code is assumed.

The CLIPS (*C*-*L*anguage *I*ntegrated *P*roduction *S*ystem) system, an expert system shell developed at NASA's Software Division Center, is used as a testbed. CLIPS consists of approximately 60 files containing over 700 functions implemented in about 30,000 lines of source code. Although it is small by commercial standards, it is a nontrivial production application which may typically be assigned to one or two software engineers.

Section 2 of this paper describes reverse engineering tools and techniques that make up the current RevEngE toolset. Section 3 outlines the reverse engineering scenarios to which we apply RevEngE. Section 4 presents the results of our analysis of the CLIPS system according to these scenarios. Section 5 summarizes the paper.

2 Tools and techniques

The current tools constituting the RevEngE toolset can be broadly classified into analysis engines, visualization interfaces, and integration mechanisms. In some cases, an individual tool may provide more than one of these functions (for example, Rigi can be used for all three). Individual tools in the RevEngE toolset communicate through a flexible object server and single global schema implemented using the Telos information model.[1]

The use of Telos as a global repository and data integration mechanism exposed a number of problems which were subsequently addressed. Most of these problems relate to data volume: although the repository is able to store large amounts of information, only small portions of this data are actually required for a given analysis. For example, certain analyses in Rigi require *some* knowledge about *many* objects, though not all information about *each* object. Quite often, complete objects are simply too big to accomplish the desired analyses. Also, some tools have limitations on the amount of information they can hold in their own workspaces. These workspace memory problems were not adequately addressed by the original built-in data retrieval operations supplied by the server, whereby only complete objects were retrievable.

These requirements drove the inclusion of appropriate data retrieval functionality in the Telos server. In particular, the following retrieval operations are now supported:

- retrieval of complete objects,
- retrieval of all objects belonging to a given class (taking account of inheritance relations),
- retrieval of partial objects (with some set of attribute categories suppressed), and
- retrieval of all information relating to a given (set of) attribute category(ies).

Corresponding data update commands are also available. The partial retrieval and update measures provide better support for the different approaches to data manipulation used by each tool in the RevEngE toolset.

[1] As described in Section 4.5, tools can also communicate using another mechanism, bypassing the Telos repository when required.

2.1 ART

ART is a prototype textual redundancy analysis engine. It generates a set of substrings ("snips") that covers the source (that is, every character of text appears in at least one substring). A set of substrings is extracted from each local context to ensure that redundant matches are not missed due to different generation processes. Matching snips are then identified. The resultant database of raw matches is translated into a form that more concisely expresses the same information to facilitate deeper analysis. Task-specific data reduction is then performed and the results summarized. This basic process may be enhanced by doing preprocessing of the source or synthesis of partial matches from short exact matches.

2.2 Ariadne

Ariadne is a set of pattern matching, design recovery, and program analysis engines. It includes techniques that perform code localization and system clustering. Code localization is achieved by comparing feature vectors calculated for every program fragment of the system and by incorporating a dynamic programming pattern-matching algorithm to compute the best alignment between two code fragments (one considered as a model and the second as input to be matched). Clustering is achieved by analyzing global data flow and data artifacts (data types, variable names, and other informal information) shared by program fragments, suggesting the use of similar concepts. Data flow is analyzed by computing the data bindings between two code fragments. This type of analysis examines the use of global variables and variables passed by reference. Data artifacts are examined by analyzing variable names and data types defined and used in every function (*common references* analysis).

In Ariadne, source code is represented within the knowledge repository as an annotated abstract syntax tree (AST). An AST was chosen as the program representation scheme because the AST maintains all relevant information from the source level. Several fine-grained analysis algorithms (for example, slicing, control flow graph analysis, and metrics calculation) can be directly applied. Nodes in the AST represent language components (statements and expressions) and arcs represent relationships between these language components. For example, an `IF`-statement is represented as an AST node with three arcs pointing to the `condition`, the `THEN`-part, and the `ELSE`-part. During Ariadne analysis, each AST node is annotated with control and data flow information. It can be a vector of software metrics; a set of data bindings with the rest of the system; or a set of keywords, variable names, and data types. The support environment is used to analyze and store the AST and its annotations, which are computed for every expression and statement node in the AST.

2.3 Rigi

Rigi[2] is a prototype realization of the PHSE:[3] an architecture for a meta reverse engineering environment [6]. It provides a basis upon which users construct domain-specific reverse engineering environments. It is instantiated for a particular application domain by *specializing* its conceptual model, by *extending* its core functionality, and by *personalizing* its user interface.

Rigi is used as part of the RevEngE toolset primarily as a visualization interface for the data produced through analyses by other tools. However, a wide variety of analyses can also be performed using Rigi itself. This is primarily due to its scripting facility, which enables it to integrate third-party tools to perform specific reverse engineering activities, such as special-purpose graph layout, to aid program understanding.

Rigi's interface with the Telos message server requires a two-way translation mechanism between the object-oriented language of the server and Rigi's own relational language RSF (*R*igi *S*tandard *F*ormat). Attributes of Telos objects become RSF tuples in Rigi. For example, since only C functions are common to both Rigi and Ariadne, data with the `functionName` attribute category for all programming objects defined in the Ariadne AST are received from the Telos server and internally converted to RSF tuples by Rigi scripts. The result is that function objects already present in the Rigi editor receive new names that correspond to the Rigi workspace.

[2] In this paper, "Rigi" refers to version V of the Rigi environment.

[3] The acronym PHSE, pronounced "fuzzy," stands for *P*rogrammable *H*yper*S*tructure *E*ditor.

3 Scenarios

Most of the effort in software maintenance is spent examining a system's implementation to understand its behavior and to recover design information. Code analysis methods are used in the reverse engineering process to help the software engineer understand system structure, determine the scope of requested changes, and detect design information that may be lost during the system's life cycle. Fundamental issues for program understanding include:

- code representation,
- structural representation of the system (modules' interaction),
- data flow and control flow graphs,
- quality and complexity metrics,
- localization of algorithms and plans,
- identification of abstract data types and generic operations, and
- multiple view system analysis using visualization and analysis tools.

Recovery of a system's design means understanding the organization of the system in terms of modules and their interfaces. Artifacts associated with the system's design are the volume and the complexity of the interfaces, specifically, of the data interfaces [7].

Analysis tasks for reverse engineering described in this paper are the following:

- **Data-bindings analysis**: A `data binding` is a tuple `<p, q, x>` where variable `x` is defined by function `p` and used by function `q`. Variable `x` is either a global or is passed by reference to `p` and `q`.

- **Common references analysis**: A common reference is a tuple <p, q, x> where functions `p` and `q` define or use variable `x`. Variable `x` is considered a common reference if it appears with the same name and data type in both functions.

- **Similarity analysis**: Five software quality and complexity metrics have been selected, specifically for their sensitivity to different program features (data flow, control flow, fan-out) [8]. These metrics are used to identify similar code fragments and to locate error-prone code.

- **Subsystem analysis**: Common references analysis and data-binding analysis are further used for system analysis. For example, the user may compute clusters of functions that define and use at least 2 and at most 10 variables, or have at least 3 and at most 15 common references. Moreover, statistical studies have shown that these analyses may reveal error-prone modules [9].

- **Redundancy analysis**: Noting where text occurs multiple times in a large source tree can ease the analysis of cut-and-paste, preprocessing, and inter-release changes.

The next section describes in more detail the application of these techniques and how they are used to assist design recovery of CLIPS.

4 Analysis

The analyses presented in this section are an application of the RevEngE toolset to the CLIPS software system. Although individual analyses of the system by each of the tools in the toolset can provide useful information, interleaved use of the tools, integrated through the Telos message server, can produce more sophisticated analyses, and can aid better understanding of the program.

The tools in the RevEngE toolset are combined in a number of ways. ART, Ariadne, and Rigi can produce raw data from the source code directly. This data is stored in the repository and its structure can be visualized and investigated with Rigi. Analyses are presented on portions of the code that are deemed interesting, based on the techniques outlined in Section 3. Ariadne analyses of data bindings, common references, and code similarity are performed. Rigi is used to explore the results of the Ariadne analyses, as well as performing an analysis of the combined Ariadne and Rigi generated data. An ART redundancy analysis is also performed on the data generated by Ariadne and is visualized with Rigi. These analyses are made possible by the integration of the tools through the Telos data repository.

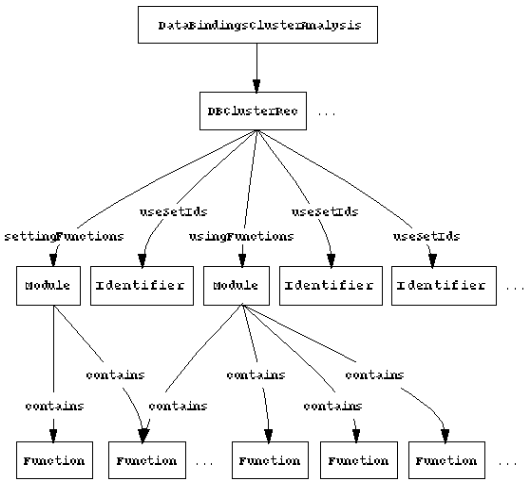

Figure 1: Ariadne data bindings

4.1 Data-binding analysis

Figure 1 shows the basic structure of the data-binding clusters. Each cluster is represented by a `DBClusterRec` object which has three attributes: `useSetIds`, `usingFunctions`, and `settingFunctions`. `useSetIds` contains identifiers that are used by the functions in `usingFunctions` and set by the functions in `settingFunctions`.

Preliminary analysis of the cluster data involved simplifications such as the removal of unconnected objects, decomposition into distinct connected components, and filtering of singly connected objects. Application of an off-line spring-layout algorithm and a further local layout revealed several meaningful clusters representing potential system modules. A view from this analysis is shown in Figure 2.

Note that the variable `memory_table` is a pivotal global, which should receive careful attention by system maintainers. Once this variable is removed, five disconnected subgraphs remain. The variables within globally referenced data types and those functions that set or use those variables are labeled and shown in columns. Those nodes without names are constructed by combining DBClusterRec and Module objects to show interconnections among Functions and Identifiers. Of the five subgraphs, two are localized in a single source file; two other subgraphs are distributed across two source files each. The largest subgraph is spread across nine source files. From this analysis of common data types, it appears that the system architecture can be viewed conceptually as consisting of five or more relatively independent subsystems. The smaller subsystems appear to be (1) a parser for inference engine initialization; (2) a parser for inference rules; (3) a parser for input source code; and (4) a test pattern generator. The inference engine itself is a member of the remaining, larger subsystem.

Another useful result from this analysis is the identification of key functions that have high coupling with the rest of the system. For example, the cluster in the lower left corner of Figure 2 consists of five functions (purge_agenda, add_activation, remove_all_activations, run, and clear_rule_from_agenda) defined in the source file engine.c. This file is one of the core modules of the expert system shell. Another example is the cluster in the top right corner. It consists of top-level functions dealing with the creation, deletion, and initialization of "facts" (a concept in the domain). This kind of analysis serves as a guiding step for identifying collections of functions from which analysis of the system can be taken further.

4.2 Common reference analysis

Another analysis that lends itself to useful subsystem decomposition is the common references cluster analysis. This type of analysis determines those identifiers that are shared among a set of functions. The results of an analysis of CLIPS are shown in Figure 3. There are four non-trivial clusters shown each having functions in the right column and the identifiers referenced by those functions in the left column. The unnamed nodes in the center column merely indicate interconnections among the functions and identifiers.

The four clusters in the figure represent sets of functions and shared variables that have small interfaces to other components in the system. The graph in the upper left corner is spread over six source files. Each of the other three graphs are localized in one or two source files. One can speculate from the data bindings and common references analyses that there are a number of weakly coupled fringe subsystems and several strongly coupled core subsystems.

The common identifiers have the same name and data type in their corresponding functions. For example, consider the graph in the upper right corner of the figure. Some of the core functions in the left column handle traversal of the object hierarchy (`PRINT_CLASSES`, `PRINT_SUBCLASSES`, `PRINT_MEMBERS`, `INPUT_A_OBJECT`), updating the object hierarchy (`save_Subclass`, `save_Member`, `read_from_file`), and updating attribute values (`getFieldsNode`, `putValueFunctionMulti`, `getValueMultiFunction`, `modifyValueFunction`). All these functions reside in the `object.c` and `method.c` modules which indeed are the ones implementing the object-oriented paradigm in CLIPS. Concluding, we may argue that common reference analysis creates clusters of functions that operate on similar concepts. These functions can be another departure point for fine-grained analysis.

4.3 Similarity analysis

Another analysis performed by Ariadne is based on metric values to detect instances of code cloning. Functions with the same structure and dataflow attributes will most likely implement similar algorithms. By examining clusters of similar functions, the developer may identify system components that deal with particular system functionality.

For example, code cloning clusters to-

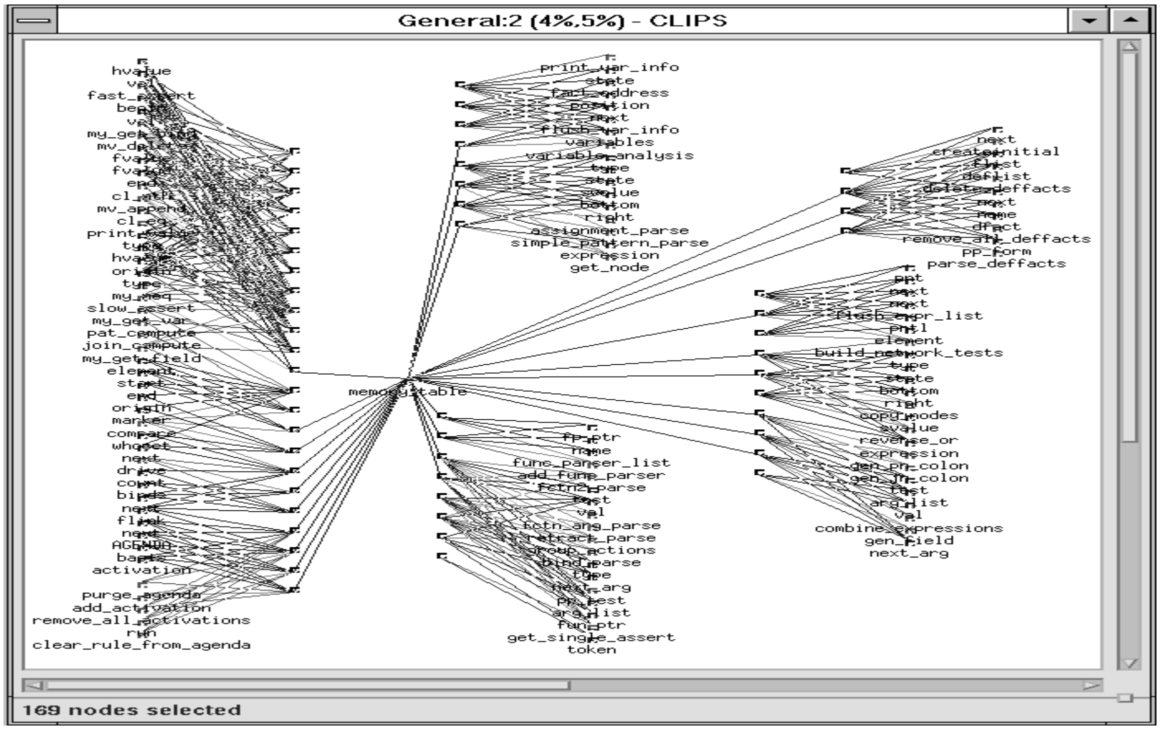

Figure 2: Preliminary view of clusters

gether several functions (`trace_off`, `trace_on`, `reset_command`, `show_requests`, and `agenda_command`) that reside in files `intrfile.c`, `intrexec.c`, `intrbrws.c`. Indeed, these modules implement the interface component of the shell.

Another example is the cluster containing functions `get_rule_num`, `get_next_rule`, `get_next_fact`, `get_next_activation`, `get_next_deffact`. These functions reside in `rulemngr.c`, `factmngr.c`, `engine.c`, `deffacts.c`, and `analysis.c`. These modules implement fact management, rule management, and the triggering of rule activation.

4.4 Subsystem analysis

The initial analysis of the combined Ariadne and Rigi data examines a small subsystem of the CLIPS program. Because the data provided by these tools is complimentary, this permits a more complete and extensive analysis than would be possible if the tools were used individually. Initially, a graph layout algorithm is used to group logically connected components of the system together. This analysis reveals two distinct subsystems, the smaller of which is shown in Figure 4.

The subsystem contains all of the component types of the larger graph on a scale small enough to analyze in detail. From the Ariadne analysis, it contains a set of six global identifiers (top left of the figure), and their associated DBClusterRec and Module nodes. From the Rigi analysis, it contains a set of ten locally referenced data types (bottom left of the figure).

Common to both the Ariadne and Rigi analyses are the Function nodes (the remaining nodes in the figure). The interface of this subsystem to the rest of the CLIPS program is through seven function calls. The interface functions that interact with the subsystem are not shown in the figure and the interfaces are represented by arcs that are only connected to one node. Five of the seven function interfaces are to an error-reporting function; therfore, they add lit-

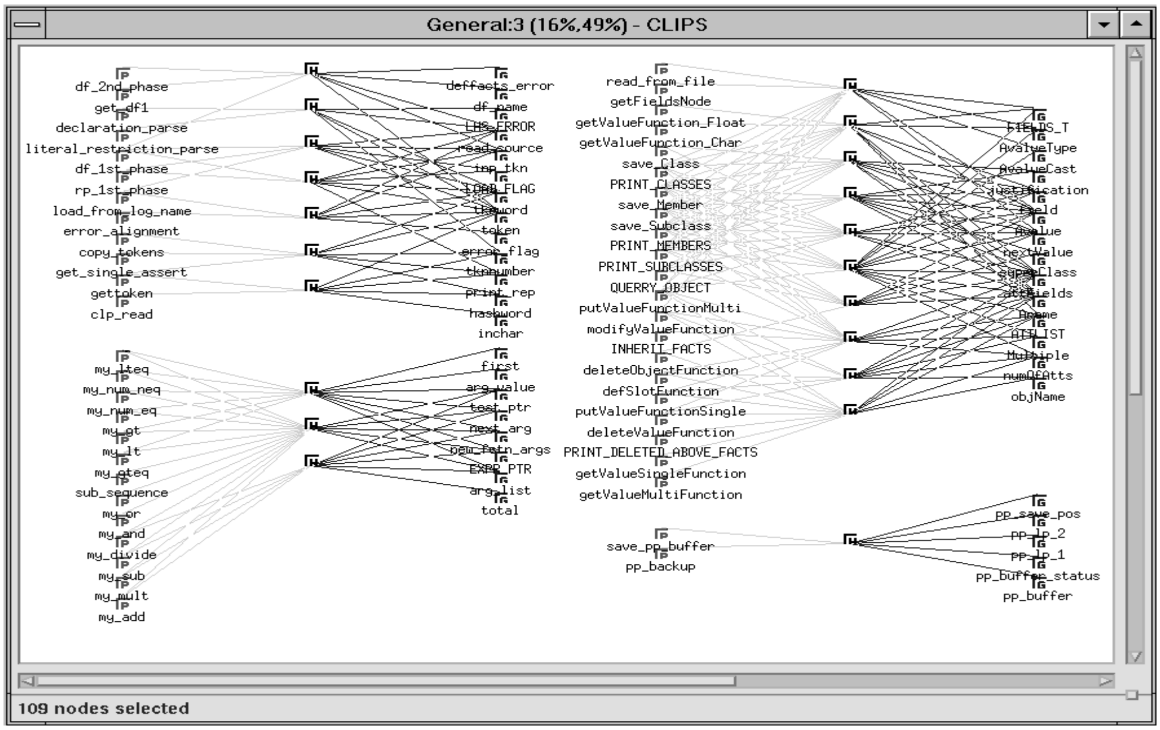

Figure 3: Common reference components

tle to our understanding of the subsystem. The other interfaces are calls to the getline function (bottom right corner of the figure), which is called by three internal functions and by two external functions. The getline function does not call any other functions in the system, and thus the subsystem has only one entry point and a large part of the subsystem seems to be unused. Looking at the graph layout provided, the lack of entry points is obvious. On further investigation of the code, it is discovered that the functions `why_command`, `how_command`, and `query_command` are invoked through function pointers.

A further analysis of the combined Ariadne and Rigi data examines the structure of the shared variables and data types of a given set of functions. Typically, a given variable is referenced by one, two, or sometimes three different `DBClusterRec` nodes. The layout algorithm applied in the previous analysis reveals a group of twelve variables with a set of associated `DBClusterRec` nodes that use five or more variables. Through the `DBClusterRec` nodes, we can find the `Module` nodes, which in turn bring us to the function nodes that reference these variables. As it turns out, these functions also reference many of the global data types that exist in the system as well. The functions are of interest because of the complex manner in which they access the set of variables. There is no other variable usage structure of similar complexity in the rest of the system.

A view from part of this analysis is given in Figure 5. In this figure, we see the objects of interest arranged in columns. From left to right we have the `Identifier` nodes, the `DBClusterRec` nodes, the `Module` nodes, the `Function` nodes, and the `DataType` nodes. There are twelve global variables that are shared by twenty-four functions. Twenty-two of the functions use the variables to control their execution and ten of the functions change the states of those variables. Changing any of the ten `set` functions can have a significant impact on the functionality of twenty-two other func-

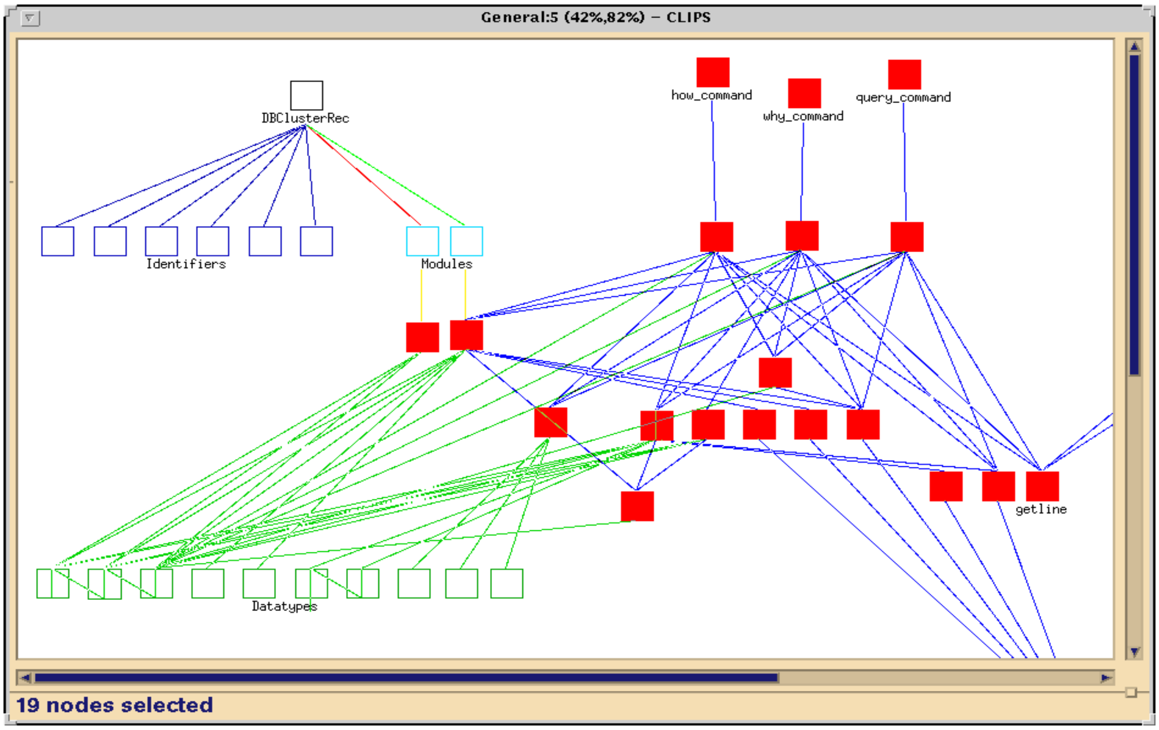

Figure 4: Subsystem analysis

tions in the system. These functions would be considered central components and, in a maintenance scenario, would likely be assigned to experienced personnel if they were to be modified. It is important to note that without the combination of the Ariadne variable data and the Rigi function call data, this potentially dangerous piece of code would not have been identified.

4.5 Redundancy analysis

This subsection illustrates how additional tools and data are integrated in the environment without requiring explicit construction of a Telos interface. This is made possible through interfacing at Rigi's relational RSF layer, already used for loose integration of layout programs that assist with Rigi visualizations. The advantage of this form of integration is primarily rapid proof-of-concept realization.

ART was applied to the CLIPS source, with exact matches of five or more lines requested. A large collection of matches were found and grouped into clusters. Four complex clusters were found and their partial order graphs calculated and laid out. The nodes and arcs—together with position information—were written into RSF tuples. The graphical display capabilities of Rigi were then used to highlight useful data and hide unessential distracting detail in the search for a better understanding of the source. Although this source does not contain a large amount of internal textual redundancy, several interesting matches were found.

A collection of 41 files (852078 characters) makes up the largest cluster. If textual redundancy is reduced by removing second and subsequent copies of repeated text, the total size is reduced by 11%. Upon inspection, using editor tools instantiated via the `fileName` attribute of `File` objects in Rigi windows, most of the matches are seen to be a result of "boiler plate" code at the beginnings of files. However, there are a number of interesting matches that stand out when the cluster graph is displayed:

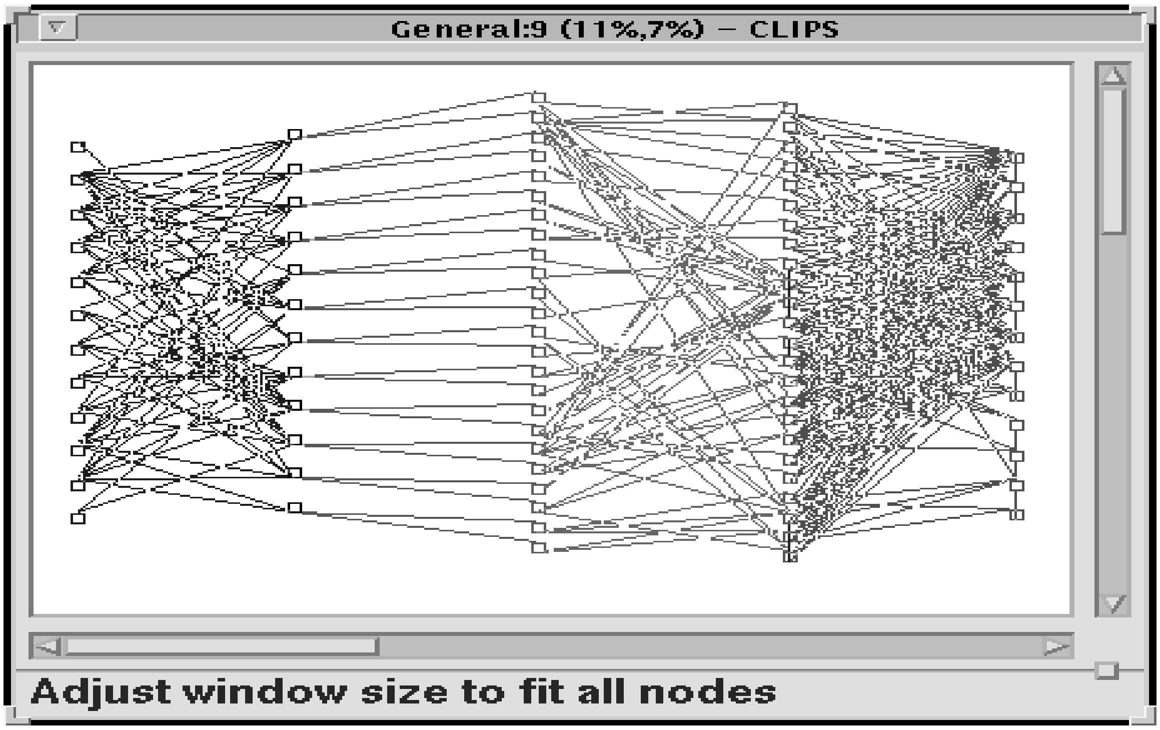

Figure 5: Shared variable analysis

- The files `match.c` and `drive.c` contain a large number of relatively small matches. It appears that one of these is partly cloned from the other. There are also a number of lines of comments in common that related to the algorithm, further supporting this contention.

- Most of the file `prinat.c` appears in the file `object.c`. It appears that `prinat.c` contains some template code that is used for constructing `object.c`.

- Most of the file `methodsFile.c` is contained in `my_methods3.c`.

- Most of the file `my_source4.c` is contained in `my_source3.c`.

The actual layout in a Rigi window for the first (and most interesting) cluster was accomplished using Rigi's programming layer and a simple, interpreted script. The layout is shown in Figure 6.

There are roughly two rows of nodes in the figure: a row of interior nodes (representing matches) located about halfway up the diagram, and a row of leaf nodes (representing files). Since the vertical coordinate measures the logarithm of the size for this layout, we can infer that the sizes of the files are nearly the same, and the sizes of the matches are also nearly the same, and are much less than the sizes of the files. To the right is a set of nodes that are not highly connected to the rest in the same way and where the file sizes are smaller and closer in size to their matches. The other matches identified above are mostly among this group of files.

Rigi proved to be useful for identifying interesting features. However, the layout algorithm used in this study requires some improvement; the treatment of very large clusters will be improved using a number of strategies. In addition, a Telos schema is being developed that will associate this file-level textual analysis with the other analyses being done.

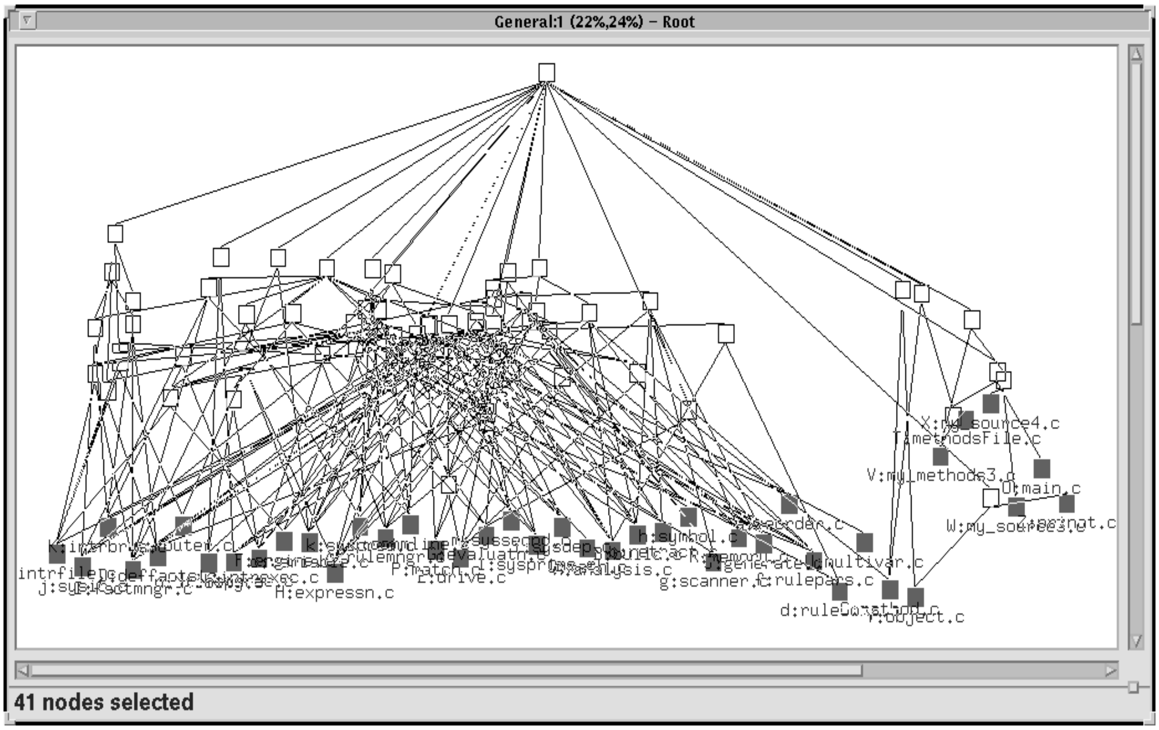

Figure 6: Rigi view of ART redundancy analysis data

5 Summary

This paper related initial experiments in applying openly integrated program understanding technologies to a single test system. Data integration is available at two levels: that of the knowledge base and message server, and that of Rigi's relational integration mechanism and scripting layer. The former requires a well-designed global schema describing shared data, while the latter permits rapid prototyping involving previously unmodeled data.

Through the design of the common Telos schema, plus the ability to "merge" data sets via end-user programming, integration of otherwise separate data sets was achieved. This permitted new analyses not possible with any one tool alone. In particular, the variable dataflow analyses produced by Ariadne were quite useful when augmenting the Rigi call and datatype dependency graph; some decomposition criteria were reinforced with the additional knowledge, and new criteria became apparent.

These initial experiments have uncovered many potential avenues for improvement. One especially noticeable problem, revealed when Rigi data and Ariadne data were integrated, was that relationships between data type objects (provided by Rigi) and variable objects (provided by Ariadne) were conspicuously absent. We intend to adjust the RevEngE schema to accommodate these links to utilize the new information.

Rigi's scripting layer provides an excellent vehicle for a certain degree of both control and data integration. However, achieving truly useful data integration at the level of the knowledge base and message-passing layer has proven to be a challenging task. Each tool views its data differently, and it is not easy to design a schema that is both complex enough to cover the data needs of all tools, yet simple enough to use efficiently and effectively.

One idea for alleviating this problem is to exploit the ability of data-consuming tools, like the Rigi editor, to load domain information stored in the knowledge base, when such an ability exists. Rigi is currently undergoing enhancements that will permit this.

Acknowledgments

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada, the IBM Software Solutions Toronto Laboratory Centre for Advanced Studies, the Information Technology Research Centre of Ontario, and the U.S. Department of Defense.

About the Authors

Morris Bernstein *School of Computer Science, McGill University, 3480 University Street, Room 318, Montréal, Québec, Canada H3A 2A7.* `zaphod@cs.mcgill.ca`. Mr. Bernstein received his B.Sc. and M.Sc degrees from McGill University. His research interests include software development, program understanding, compiler design, and application-domain languages. He is currently a research assistant with primary responsibility for the McGill portion of the RevEngE project.

Brian Corrie *Department of Computer Science, University of Victoria, P.O. Box 3055, Victoria, BC, Canada V8W 3P6.* `bcorrie@csr.uvic.ca`. Mr. Corrie is employed as a Research Associate in the Rigi Group at the University of Victoria. He received his B.Sc. and M.Sc. degrees in Computer Science from there in 1988 and 1990, respectively. His primary research interests include scientific visualization, parallel computer systems for graphics, and scientific computing. He is a member of IEEE Computer Society.

J. Howard Johnson *Institute for Information Technology, National Research Council Canada, Montreal Road, Building M-50, Ottawa, Ontario, Canada K1A 0R6.* `johnson@iit.nrc.ca`). Dr. Johnson is a Senior Research Officer with the Software Engineering Laboratory of the National Research Council. His current research interest is Software Re-engineering and Design Recovery using full-text approaches. He received his B.Math. and M.Math. in Statistics from the University of Waterloo in 1973 and 1974 respectively. After working as a Survey Methodologist at Statistics Canada for four years, he returned to the University of Waterloo and in 1983 completed a Ph.D. in Computer Science on applications of finite state transducers. Since then, he has been an assistant professor at the University of Waterloo and later a manager of a software development team at Statistics Canada before joining NRC.

Kostas Kontogiannis *School of Computer Science, McGill University, 3480 University Street, Room 318, Montréal, Québec, Canada H3A 2A7.* `kostas@binkley.cs.mcgill.ca`. Mr. Kontogiannis received a B.Sc degree in Mathematics from the University of Patras, Greece, and an M.Sc degree in Artificial Intelligence from Katholieke Universiteit Leuven in Belgium. Currently, he is a Ph.D candidate in the School of Computer Science at McGill University. His interests include plan localization algorithms, software metrics, artificial intelligence, and expert systems.

James G. McDaniel *Department of Computer Science, University of Victoria, P.O. Box 3055, Victoria, BC, Canada V8W 3P6.* `jmcdanie@csr.uvic.ca`. Dr. McDaniel received B.Sc degree in Mechanical Engineering at Case Western Reserve University, Cleveland, Ohio, an M.Sc in Electrical Engineering at Cornell University, Ithaca, New York, and a Ph.D in Computer Science and Health Information Science at the University of Victoria, B.C. He has worked in the area of commercial software consulting and management for fifteen years. Currently he is a Research Associate for the Rigi Project at the University of Victoria. His interests include software engineering, reverse engineering, and networks. He is a member of the IEEE, ACM, and COACH societies.

Ettore Merlo *Département de Génie Électrique (DGEGI), École Polytechnique de Montréal, C.P. 6079, Succ. Centre Ville, Montréal, Québec, Canada H3C 3A7.* `merlo@rgl.polymtl.ca`. Dr. Merlo graduated from the University of Turin, Italy, in 1983 and obtained a Ph.D. degree in computer science from McGill University in 1989. From 1989 until 1993, he was the lead researcher of the software engineering group at the Computer Research Institute of Montreal. He is currently an assistant professor of computer engineering at École Polytechnique de Montréal, where his research interests include software re-engineering, software analysis, and artificial intelligence. He is a member of IEEE Computer Society.

Renato De Mori *School of Computer Science, McGill University, 3480 University Street, Room 318, Montréal, Québec, Canada H3A 2A7.* `demori@cs.mcgill.ca`. Dr. De Mori received a doctorate degree in Electronic Engineering from Politecnico di Torino, Italy, in 1967. Since 1986, he has been a professor and the director of the School of Computer Science at McGill University. In 1991, he became an associate of the Canadian Institute for Advanced Research and project leader of the Institute for Robotics and Intelligent Systems, a Canadian Centre of Excellence. His current research interests are stochastic parsing techniques, automatic speech understanding, connectionist models, and reverse engineering. He is the author of many publications in the areas of computer systems, pattern recognition, artificial intelligence, and connectionist models. He is on the board of the following international journals: the *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *Signal Processing*, *Speech Communication*, *Pattern Recognition Letters*, *Computer Speech*, and *Language and Computational Intelligence*. He is a fellow of the IEEE Computer Society.

Hausi A. Müller *Department of Computer Science, University of Victoria, P.O. Box 3055, Victoria, BC, Canada V8W 3P6.* `hausi@csr.uvic.ca`. Dr. Müller is an associate professor in the Department of Computer Science at the University of Victoria, where he has been since 1986. He received his Ph.D. in computer science from Rice University in 1986. From mid 1992 to mid 1993, he was on sabbatical at CAS, working in the program understanding project. His research interests include software engineering, software analysis, reverse engineering, re-engineering, programming-in-the-large, software metrics, and computational geometry. He is currently a Program Co-Chair of the *International Conference on Software Maintenance* (ICSM '94) in Victoria and the *International Workshop on Computer Aided Software Engineering* (CASE '95) in Toronto. He is a member of the editorial board of *IEEE Transactions on Software Engineering*.

John Mylopoulos *Department of Computer Science, University of Toronto, 6 King's College Road, Toronto, Ontario, Canada M5S 1A4.* `jm@ai.utoronto.ca`. Dr. Mylopoulos is a professor of computer science at the University of Toronto. He received his Ph.D. degree from Princeton University in 1970. His research interests include knowledge representation and conceptual modeling, covering languages, implementation techniques for large knowledge bases, and the application of knowledge bases to software repositories. He is currently leading a number of research projects and is principal investigator of both a national and a provincial Centre of Excellence for Information Technology. His publication list includes more than 120 refereed journal and conference proceedings papers and three edited books. He is the recipient of the first ever Outstanding Services Award given out by the Canadian AI Society (1992), and also a co-recipient of a best paper award at the *16th International Conference on Software Engineering*.

Martin Stanley *Department of Computer Science, University of Toronto, 6 King's College Road, Toronto, Ontario, Canada M5S 1A4.* `mts@ai.utoronto.ca`. Mr. Stanley received his M.Sc. degree in computer science from the University of Toronto in 1987. His research interests include knowledge representation and conceptual modeling, with particular application to the building of software repositories. He is currently a research associate in the Department of Computer Science at the University of Toronto, with primary responsibility for the Toronto portion of the RevEngE project.

Scott R. Tilley *Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213-3890.* `stilley@sei.cmu.edu`. Dr. Tilley is a Member of the Technical Staff at the SEI. He received the Ph.D. degree in Computer Science from the University of Victoria in 1995. His first book on home computing was published in 1993. His research interests include end-user programming, hypertext, program understanding, reverse engineering, and user interfaces. He is currently part of the Disciplined Engineering Program's Reengineering Center. He is a member of the ACM and the IEEE.

Michael Whitney *Department of Computer Science, University of Victoria, P.O. Box 3055, Victoria, BC, Canada V8W 3P6.* `mwhitney@csr.uvic.ca`. Mike Whitney works as a Research Assistant at the University of Victoria, from whence he received his M.Sc. and Ph.D. degrees in 1988 and 1993 respectively. He is not a member of any Communities, Societies, or Associations.

Kenny Wong *Department of Computer Science, University of Victoria, P.O. Box 3055, Victoria, BC, Canada V8W 3P6.* `kenw@csr.uvic.ca`. Mr. Wong is a Ph.D. candidate in the Department of Computer Science at the University of Victoria. He worked in the program understanding project while at CAS during the summer of 1993 and 1994. His research interests include program understanding, runtime analysis, user interfaces, object-oriented programming, and software design. He is a member of the ACM, USENIX, and the Planetary Society.

References

[1] J. Mylopoulos, M. Stanley, K. Wong, M. Bernstein, R. D. Mori, G. Ewart, K. K. amd Ettore Merlo, H. Müller, S. Tilley, and M. Tomic. Towards an integrated toolset for program understanding. In *Proceedings of the 1994 IBM CAS Conference (CASCON '94),* (Toronto, ON; October 31 - November 3, 1994), pages 19–31, November 1994.

[2] K. Kontogiannis, R. DeMori, M. Bernstein, and E. Merlo. Localization of design concepts in legacy systems. In H. A. Müller and M. Georges, editors, *Proceedings of the International Conference on Software Maintenance* (Victoria, B.C., Canada; September 19-23, 1994), pages 414–423, 1994.

[3] H. Johnson. Visualizing textual redundancy in legacy code. In *Proceedings of the 1994 IBM CAS Conference (CASCON '94),* (Toronto, ON; October 31 - November 3, 1994), pages 9–18, November 1994.

[4] S. R. Tilley, K. Wong, M.-A. D. Storey, and H. A. Müller. Programmable reverse engineering. *International Journal of Software Engineering and Knowledge Engineering*, 4(4):501–520, December 1994.

[5] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: Representing knowledge about information systems. *ACM Transactions on Information Systems*, 8(4):325–362, October 1990.

[6] S. R. Tilley. *Domain-Retargetable Reverse Engineering.* PhD thesis, Department of Computer Science, University of Victoria, January 1995. Available as technical report DCS-234-IR.

[7] T. DeMarco. *Controlling Software Projects: Management, Measurements, and Estimation.* Yourdon Press, 1986.

[8] R. Adamon. Literature review on software metrics. Technical report, ETH Zürich, 1987.

[9] R. W. Selby and V. R. Basili. Analysing error-prone system structure. *IEEE Transactions on Software Engineering*, 17(2):141–152, February 1991.