On Inserting Program Understanding Technology into the Software Change Process

Kenny Wong
Department of Computer Science
University of Victoria
Victoria, BC, Canada

Abstract

Program understanding technologies can be applied effectively in the analysis phase of a software change process. The analysis phase naturally follows a goal-driven metaprocess. Described are issues involved with inserting program understanding technology into existing practice and into such a metaprocess. The implied processes of program understanding and reverse engineering tools play an important role. These issues pose major problems for the acceptance of redocumentation tools such as Rigi, an evolvable reverse engineering tool. An example using Rigi and its analysis methodology for change-impact analysis is considered.

Keywords: program understanding, reverse engineering, software process, metaprocess, technology insertion.

1 Introduction

The importance of software evolution has increased now that the focus of the software industry has shifted from completely new software development to long-term maintenance. The expense in existing software is considerable, and it is not always possible to scrap it and start over. The effective maintenance of this code is critical for keeping up with changing needs. As a result, the software change process and the role of analysis within that process have become increasingly crucial for evolving quality software.

1.1 Evolution

The fundamental business goals that drive the evolution of a software product [1] are the needs to:

- maximize customer satisfaction,
- minimize effort and cost, and
- minimize defects.

Software analysis and program understanding play a large role in achieving these goals throughout the lifetime of the product. The software is ultimately tied to the objectives of the business.

Software changes to address these three goals can be categorized into three loosely-corresponding types:

- adaptive changes,
- perfective changes, and
- corrective changes.

Adaptive changes address customer needs and accommodate technological improvements. Perfective changes seek to make future evolution somehow better, more manageable, and less costly. Corrective changes focus on detecting, tracking, and diagnosing defects and their root causes. The effective management and execution of these changes is critical.

1.2 Program understanding

The type of change greatly influences the selection of appropriate program understanding strategies. For example, the tracking and categorization of defects are an important part of the change process. Finding defects in the source code involves diverse program understanding technologies such as defect filtering [2], structural redocumentation [3], pattern matching at various levels of abstraction, and run-time analysis. Effective understanding is also necessary for factoring and optimizing code, porting to different platforms, exploiting reusable components, and adding new features. Applications such as medical instrumentation and process control require a level of quality that is only possible with a thorough understanding of the software.

Because of changes during evolution, the software architecture often degrades in that the code no longer follows the original design criteria. This is especially true for legacy software systems, which are typically large, complex, poorly structured, and resist change. Some software systems are badly written from the start because of the enormous pressure on developers to ship the product out the door, even if the bugs and unsound structure later cause maintenance nightmares. Somebody still has to understand these high entropy systems. Thus, program understanding is an important part of software preservation.

1.3 Software change process

Changes generally arise by reviewing customer needs for new features and considering requests for fixes. The rationale for these changes ultimately points back to the fundamental business goals. Each change leads to a change process where it is typically planned (scheduled), studied, developed, inspected, and tested. Program understanding can play an effective role in analyzing changes. Certain analysis results can then be used as input into followup code modification, inspection, and metrics gathering activities.

The analysis must consider and answer several direct questions:

- what is the change,
- why is the change needed,
- whether to proceed with the change,
- where to place the change,
- how to carry out the change,
- when to perform the change, and
- who should perform the change.

An answer to the whether question entails the costs, benefits, impact, and feasibility of the change. This provides a deeper analysis of the change beyond the initial review (what, why) and may result in further decomposing the change into smaller pieces or even postponing it. We might have tools to answer the where question, programmers to answer the how question, and managers to decide the when, who, and whether questions. Consequently, the analysis itself involves multiple agents and/or roles, and has a major effect on the organization of the business.

1.4 Reverse engineering

An understanding of the existing software, the implementation context of the change, may be obtained through a reverse engineering approach. This approach provides computer assistance, often using compiler technologies (lexical, syntactic, and semantic analysis). Typically, this consists of an automatic parsing phase followed by a more intensive discovery phase that uses diverse pattern matching techniques at textual, syntactic, structural, functional, or behavioral levels [4].

The process of reverse engineering identifies software building blocks, extracts dependencies, produces higher-level abstractions to manage complexity, and presents pertinent summaries. One application is to produce architectural views of the large-scale structure of the software. A classical use of these views is to redocument an existing software system whose documentation is lost or lacking. The end product is better documentation and understanding, helping to guide engineers to making the change properly.

2 Analysis

This section considers the software analysis process and requirements necessary for introducing program understanding technology into the process. The metaprocess behind an analysis process is discussed since it has important implications for the success of adopting program understanding tools. Desirable features of tools to support effective analysis are also described.

2.1 Goal-directed metaprocess

The complex nature of software analysis and program understanding suggests a goal-directed approach oriented to actual needs (and the circumstances that arise) instead of a technology-oriented approach on perceived needs. That is, program understanding technologies such as slicing [5], structural redocumentation, type inference, pattern matching, defect filtering, and program visualization need to be inserted within the context of actual needs and ultimately business objectives.

A goal-directed strategy or metaprocess is more task-oriented and tends to avoid the situation of a solution looking for a problem. The proposed user-assisted strategy is based on a paradigm of goal, question, analysis, tool, and action. This paradigm is inspired by [6] and GQM [7], with the extension of integrating tool selection. The paradigm guides the analyst from business, quality-related, or project-specific goals and subgoals, through specific questions, through relevant program understanding analyses, through appropriate tools and features, to recommended actions for achieving the goals.

Typically, a goal leads to finer subgoals and a tree of questions, analyses, tools, and actions. A concrete goal-to-action path is called a scenario and represents a distinct task. An experienced analyst generally passes through multiple, related, possibly prioritized scenarios for analyzing a request and achieving a particular goal. Each program understanding tool may have several suggested scenarios of usage. The compilation of such scenarios forms a useful, high-level framework for the introduction of program understanding tools and techniques to organizations as well as individual developers and managers. These scenarios also need to be publicized, reused, and evolved. The following are several example scenarios.

For a corrective request to fix memory usage problems:

- **goal:** minimize defects
- **question:** where to place the changes
- **analysis:** run-time memory usage analysis
- **tool:** heap debugger
- **action:** perform changes according to discovered memory defects

For a corrective request to find coding standards violations:

- **goal:** minimize defects
- **question:** what are the changes
- **analysis:** defect filtering on coding standards violations
- **tool:** Software Refinery with defect filtering package
- **action:** categorize and prioritize discovered defects

For a perfective request to reduce code size:

- **goal:** minimize maintenance effort
- **question:** where to place the changes
- **analysis:** textual redundancy analysis of cut-paste operations
- **tool:** tool for analysis of redundancy in text [8]
- **action:** factor code according to significant redundancies

For a perfective request to better manage code complexity:

- **goal:** minimize maintenance effort
- **question:** who to perform changes
- **analysis:** graph complexity of module interconnections
- **tool:** Rigi central and fringe components
- **action:** assign senior personnel to maintain central components and junior personnel to maintain fringe components

For an adaptive change request to add a feature:

- **goal:** maximize customer satisfaction
- **question:** where to place the change
- **analysis:** structural redocumentation
- **tool:** Rigi views
- **action:** consult architectural view for appropriate location

Scenarios may be organized along at least five dimensions to help determine which are applicable in a particular context.

- scope (global versus local),
- granularity (large versus small),
- abstraction level (textual, syntactic, structural, functional, behavioral, application),
- domain knowledge (implementation versus application, general versus specific),
- automation level (manual, semi-automatic, fully automatic).

The goal-directed paradigm is an opportunistic, short-sighted, but "real-world" strategy. Tools and techniques are only invoked on a lazy, as-needed basis to answer some question and achieve some goal. In general, this strategy is natural as the understanding process itself is opportunistic, iterative, and piecemeal. Analysts make intelligent guesses, steer computations on-the-fly, gather many different pieces of information, and then integrate the pieces in some sensible way. Mechanisms and infrastructures become less important than immediate results. Tools have to be effective within the normal flow of real-world development.

However, some tools and techniques need a preprocessing or priming stage that does not relate directly to a goal. For example, a reverse engineering tool may need to parse the source code into some program representation (or repository) that can more easily be queried and analyzed. Some work products, like documentation, are useful mostly for presumed future needs. For example, a view of detected defects by category is useful to future inspection activities or future analyses where a given defect needs to be compared with prior ones. Such products are typically produced on an eager basis with the intention that they benefit the future (sometimes more so than the immediate present). Housekeeping, documentation, and support tasks that are part of the tool's methodology are outside the strict goal-directed paradigm. Such tasks benefit more in the long term, making program understanding technology insertion into the short-term, results-driven change process more difficult. Thus, there is an interplay of lazy and eager strategies to consider when adopting tools.

2.2 Evolving the process

Software change processes may evolve in three major components [9]:

- the software production process,
- its metaprocess, and
- the corresponding process support.

The evolution may occur in each component in isolation or together mutually. The change production process may evolve through new tools and techniques. The metaprocess may evolve by a refinement in strategy that maintains the same production process. Process support may be replaced, extended, or refined as well.

New tools, techniques, and processes often cannot be inserted into the existing change process without further preparation. A study is required of current analysis needs. The selection of new tools and techniques requires much evaluation and a metaprocess in itself. Subsequently, we must prepare the new target process, understand the inherent methodologies assumed by the approved tools and techniques, and define a process for the transition. The transition may involve user training, motivation, data conversions, new terminology, and changes in operating procedures. These activities need to be enumerated and planned so that they can be budgeted and measured in terms of cost and status of progress. The benefits of the new technology has to be gauged against the cost of this transition. One strategy for measuring the effect of tool insertion is the Method for Planned Tool Insertion described in [10].

2.3 User needs

Naturally, tool users have high expectations and want guarantees. In particular, they want tools that:

- bring results within the current release cycle,
- offer industrial strength robustness and scalability,
- come with long-term support,
- can be integrated into the normal flow of development,
- provide evidence of success in other industrial projects,
- present a virtually zero learning curve, and
- have low overhead.

Tools that focus mostly on the future may have difficulty helping the analyst within the current release cycle. Some of these expectations (like robustness) are very difficult to attain, especially for tools that are research prototypes. In some sense, tools may need to become more specialized (or adaptable) to do a few things very well. Tools may also need to be more lightweight in terms of methodology to afford easier insertion into the current development environment. These needs will depend on how much the users want to adapt themselves to the methodology. In general, the implied process or methodology of a tool must somehow coexist with (and not impose on) the goal-directed analysis process.

2.4 Tool requirements

Tool developers should not dwell on only the immediate, short-term needs of the users. There are several desirable features of the tools themselves to make them more effective at:

- codifying analysis experience,
- capturing and exploiting historical information from previous analyses,
- exploring what-if cases,
- providing useful documentation, and
- easing manageability

These features are primarily beneficial in the long term for future analyses.

Program understanding activities can be somewhat chaotic, with analysts making guesses, questions, and actions using their own experiences, heuristics, and judgment. These activities need to become more mature and systematic, but this is often difficult to achieve. Moreover, the sequencing of guesses and applied heuristics are often not captured. These difficulties may stem from the problem that heuristics are often very domain-specific, whereas mature and systematic techniques tend to be quite general.

One partial solution for capturing this domain-specific experience is to write conceptual models to represent the data and write process scripts or programs to codify the tasks [11]. These scripts enact a process within the larger analysis process and may be recorded automatically for quicker process prototyping. The gathering and management of these scripts forms important historical information for future analyses in the same domain. The goal-driven framework of scenarios can be augmented with script-driven, domain-specific heuristics to help improve program understanding practice. Moreover, the framework needs to be annotated with reasons why a question or heuristic is applicable.

Trial and error in program understanding may produce a flood of work products because different inputs, cases, or analysts. These products include data files, models, scripts, annotations, and views. A particular view may be the result of a particular model, but not others. A model may be a refinement of a past model. The analysis may need to be rolled back to a prior stage. The examination of past and partial work products may be useful toward capturing experience for future analyses. What is needed is some form of version management that supports such what-if analyses. Essentially, we need bookmarks in time as well as space (that is, support for a kind of electronic lab notebook for the program understanding analyst [12]).

3 Rigi

This section describes experiences with a specific tool called Rigi. Rigi is a software environment under development at the University of Victoria for program understanding. One aim is to extract abstractions from software representations and transfer this information into the minds of software engineers. The most recent results of the Rigi project include:

- an interactive graph editor,
- a reverse engineering methodology,
- measures for evaluating the quality of structural abstractions [13],
- a documentation strategy using views [14], and
- an extension and integration mechanism via a scripting language [15].

This environment can help answer some of the questions that arise in the analysis parts of the software change process.

3.1 Methodology

The reverse engineering methodology used when applying Rigi has evolved over the years and now consists of several stages:

- conceptual modeling,
- feature extraction,
- subsystem composition,
- graph manipulation and analysis,
- view creation,
- script writing, and
- user-interface customization.

Conceptual modeling involves defining the types of nodes, arcs, and attributes that correspond to the artifacts in the subject software system and concepts in the application domain. This definition is used by the graph editor, customizing it as part of a tool evolution approach to reverse engineering.

Feature extraction is initially automatic and involves parsing the source code of the subject system into a stream of tuples representing software artifacts. This produces a flat resource-flow graph of the software. To manage the complexity, the extraction phase is followed by a largely semi-automatic one that exploits human pattern recognition skills to identify and compose subsystems.

Subsystem composition is a recursive process whereby software building blocks such as data types, procedures, and subsystems are clustered into composite subsystems [16]. This builds multiple, layered hierarchies as higher-level abstractions that reduce the complexity of understanding large software systems. The criteria for what comprises a subsystem depends on the analysis goal, audience, and application domain. For example, a subsystem may represent an abstract data type, personnel assignment, defect category, or any clustering concept.

The resulting software hierarchies can be manipulated, visualized, and navigated with the graph editor. One possible use is to expose properties and anomalies of the software structure for managing risk and deciding personnel assignments. For example, highly complex, central components are best handled by the senior maintainers. As a form of "conceptual slicing", an exact interface analysis computes the dependencies to, from, and within a subsystem at any level in the hierarchy;

A Rigi view is a snapshot or bookmark that reflects the spatial state of the graph model and the visual state of the user interface. In particular, a view is a reloadable bundle of visual and textual frames that contain, for example, call graphs, overviews, projections, exact interfaces, reports, and annotations [17]. Each view reconstitutes a particular perspective to highlight maintenance constraints and problems. The focus is to construct readable, accurate, and up-to-date documentation about the subject system.

Rigi offers a scripting language that allows analysts to codify reverse engineering activities. The analyst can complement the built-in operations with external algorithms for graph layout, metrics, software analysis, etc. Complex analysis tasks can be automated for more consistency and repeatability. The user interface is customizable in that menus and dialogs can be added or modified through commands in the language [18]. Script writing and user-interface customization evolve the tool to better analyze the subject, as part of prototyping and optimizing processes in the methodology.

3.2 Tool insertion

When inserting Rigi into a goal-directed analysis paradigm, we must consider the implied methodology of the tool. Rigi has three major preparatory tasks: conceptual modeling, feature extraction, and subsystem composition. Since these prerequisite tasks are potentially quite involved, they need to be prepared fully before any particular goal-directed analysis takes place. The subsystem composition stage is generally the most intensive and least able to be automated. Even when subsystem composition can be partially automated, someone still has to design, write, and test the controlling scripts. The high overhead of this stage poses a major problem for adopting Rigi in industrial settings where time is often limited. Rigi also has three major "forward thinking" tasks that tend to ease future uses of the tool or future analyses. These tasks are view creation (redocumentation), scripting writing, and user-interface customization. Of all the stages in the methodology, graph manipulation and analysis best conforms to the goal-directed paradigm. This stage is highly interactive, and in our experience, it seems the most readily understood and exploitable by analysts that need specifics (such as module dependencies).

One idea to increase the acceptance of Rigi may be to have an expert take care of supporting tasks, such as:

- modifying schemas in the conceptual model,
- looking at previous change analyses to write useful scripts,
- maintaining required views,
- supporting other analysts with particular program understanding needs,
- evolving Rigi to new applications, and
- demonstrating program understanding capabilities.

Realistically, having such a local expert to champion the technology is likely more effective than training and motivating every analyst in adapting and/or operating Rigi themselves. The expert can tailor the Rigi tool with specific conceptual models and scripts for use by the other analysts. In the past, we have sent our own Rigi developers to be these experts, often as part of a pilot project where Rigi has to prove itself on some non-trivial piece of industrial software. Generally, this seems only fruitful if the developer can consult for at least a few months to learn some domain-specific knowledge about the subject software. Moreover, it is vital that at least one analyst at the organization spends a significant and successful time using Rigi. Otherwise, once the developer leaves, the tool remains unadopted.

4 Example

This section describes an example scenario which uses Rigi. It also indicates the amount of effort needed to answer an analysis question; fortunately, much of this work is reusable for future analyses. There is a corrective change request to fix a defect into a particular component:

- **goal:** minimize defects
- **question:** whether to perform the change
- **analysis:** dependency analysis
- **tool:** Rigi exact interfaces
- **action:** perform change if no external components are affected

The analysis determines the impact of changes to other components should the given component be modified. The extent or scope of this impact is useful information for assigning personnel and estimating the effort to fix the defect. Prior to the analysis, there is a preparation phase. Following the analysis, there is a forward-feeding phase that benefits future analyses of the same kind.

4.1 Preparation

The preparation involves: conceptual modeling, feature extraction, and subsystem composition. The subject system is a small list processing program, written in C, which uses a list data type and an auxiliary element data type. The particular C conceptual model represents functions and data types, with call or data access relationships among them. The specification of the model is through a file of triplets that is loaded into the graph editor. The model is stratified into six increasingly domain-specific levels, among three layers:

- metaclass (modeling axioms, attributed graphs)
- upper class (Rigi graph model)
- middle class (Rigi concepts)
- lower class (C concepts)
- upper token (default values)
- lower token (actual subject data)

Each triplet represents a verb, noun, and object. The *in* verb specifies an instance-of inheritance relationship between metaclass, class, and token layers. The *isa* verb specifies an is-a inheritance relationship within a layer. The modeling concepts are similar to Telos [19].

Here is part of the class layer:

```
# upper class level -----
in RigiNode Node
in RigiArc Arc
in RigiAttr Attr

# declare visible node/arc attrs
isa Label RigiAttr
isa Color RigiAttr
isa Icon RigiAttr
isa Position RigiAttr
isa Width RigiAttr

# attach types of node attrs
Label RigiNode String
Color RigiNode String
Icon RigiNode String
Position RigiNode String

# attach types of arc attrs
Label RigiArc String
Color RigiArc String
Width RigiArc Integer

# middle class level -----
isa level RigiArc
level RigiNode RigiNode

# lower class level -----
isa declarable RigiNode
isa type declarable
# C struct
isa Data type
# C function
isa Module declarable

isa call RigiArc
call Module Module
isa data RigiArc
data Module Data
data Data Data
```

The feature extraction is done through a C parser and generates triplets that comprise the lower token level of the model for the subject program.

The subsystem composition phase is based on the criteria of abstract data types, identifying both the list and element data types and their corresponding access functions. These data types and access functions are collapsed into an `ADT_list` subsystem and an `ADT_element` subsystem (Figure 1).

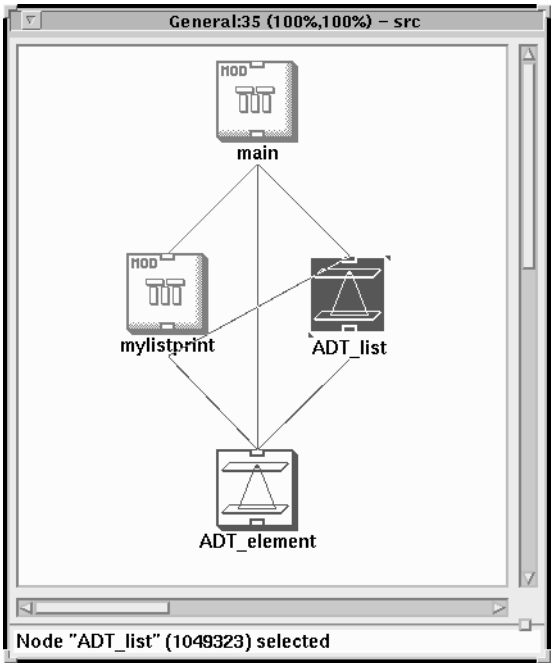

Figure 1: Abstract data type subsystems.

4.2 Graph analysis

An exact interface or dependency analysis report can be produced on demand for a selected subsystem (containing the defect in question). The report provides three kinds of information:

- provisions,
- requirements, and
- internalizations.

A provision is a dependency from a node inside the subsystem to a node outside the subsystem. A requirement is a dependency from a node outside the subsystem to a node inside the subsystem. An internalization is a dependency between two nodes inside the subsystem.

Here is the report on the `ADT_list` subsystem:

```
Exact Interface for
subsystem ADT_list:

Provisions = 4

listcreate provides 1 object.
listfirst  provides 1 object.
listinsert provides 1 object.
listnext   provides 1 object.

listcreate -> main(src)
```

```
listfirst  -> mylistprint(src)
listinsert -> main(src)
listnext   -> mylistprint(src)

Requirements = 3

list       requires 1 object.
listinsert requires 1 object.
listnext   requires 1 object.

list       <-
    element(ADT_element)
listinsert <-
    elementsetnext(ADT_element)
listnext   <-
    elementnext(ADT_element)

Internalizations = 6

listcreate internalizes 1 object.
listfirst  internalizes 1 object.
listid     internalizes 1 object.
listinit   internalizes 1 object.
listinsert internalizes 1 object.
listnext   internalizes 1 object.

listcreate <- list
listfirst  <- list
listid     <- list
listinit   <- list
listinsert <- list
listnext   <- list
```

4.3 Forward feeding

Although the textual report may be sufficient, future change-impact analyses may be more effective if the dependencies can be visualized. This involves some script writing to produce the visualization. The following script extracts the required report data on the selected subystem and a given set of dependencies.

```
proc slice { pA } {
  set S [rcl_select_get_list]
  set rn {}
  set pn {}
  set in {}

  foreach s $S {
    # get exact interface
    set ei \
    [typed_node_interface $s $pA]
    # arcs are returned
    set ra [lindex $ei 0]
    set pa [lindex $ei 1]
    set ia [lindex $ei 2]

    # get nodes
    foreach n $ra {
      ladd rn [rcl_get_arc_dst $n]
      ladd in [rcl_get_arc_src $n]
    }
    foreach n $pa {
      ladd pn [rcl_get_arc_src $n]
      ladd in [rcl_get_arc_dst $n]
    }
    foreach n $ia {
      ladd in [rcl_get_arc_src $n]
      ladd in [rcl_get_arc_dst $n]
    }
  }
  set common [lintersect $rn $pn]
  set rn [ldiff $rn $common]
  set pn [ldiff $pn $common]

  # get all nodes
  set an {}
  foreach n $rn {
    lappend an $n
  }
  foreach n $pn {
    lappend an $n
  }
  foreach n $common {
    lappend an $n
  }
  foreach n $in {
    lappend an $n
  }
  return [list $rn $pn \
    $in $common $an]
}
```

The following script presents the extracted information in a circular layout with four quadrants containing:

- internalizations in the selected subsystem (northwest),
- leaf nodes to and from which the subsystem provides and requires (northeast),
- leaf nodes to which the subsystem provides (southwest), and
- leaf nodes from which the subsystem requires (southeast).

If necessary, the subsystems that contain the leaf nodes can be visualized in an overview window of the subsystem hierarchy.

```
proc circle { L } {

  rcl_select_all
  rcl_open_projection

  set q(0) [lindex $L 0]
  set q(1) [lindex $L 1]
  set q(2) [lindex $L 2]
  set q(3) [lindex $L 3]

  rcl_select_none
  foreach n [lindex $L 4] {
    rcl_select_id $n 1
  }
  rcl_select_invert
  rcl_filter_selection

  rcl_select_all
  rcl_scale_none
  rcl_set_scale_factor 25
  rcl_scale_by_factor
  # got all the nodes involved

  set a 500
  set b 500
  set pi 3.1415926

  for {set i 0} {$i < 4} \
  {incr i} {
    set d [expr ($pi / 2) / \
      ([llength $q($i)] + 2)]
    set t [expr $i * $pi / 2]

    foreach n $q($i) {
      rcl_set_node_position $n \
        [expr round($b * cos($t)) + $b] \
        [expr round($a * sin($t)) + $a]
      set t [expr $t + $d]
    }
  }
  rcl_refresh
}
```

Using the diagram, dependencies to and from the subsystem and procedure(s) containing the defect can be seen (Figure 2).

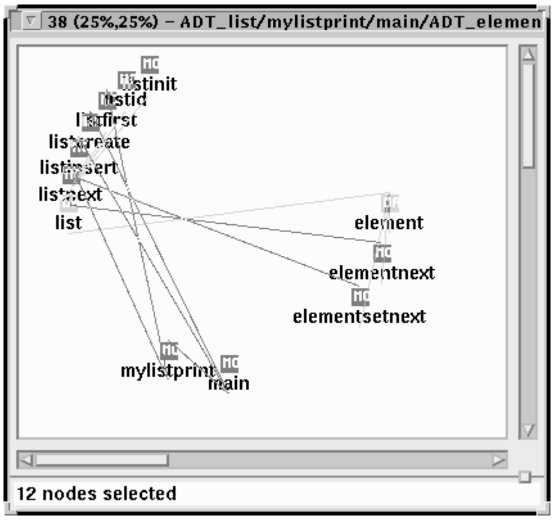

Figure 2: Circle layout of dependencies.

5 Summary

Program understanding tools and techniques can play an important role in the analysis part of the software change process. In particular, tools such as Rigi can use exact interfaces to help answer the question of whether certain changes should proceed. It is important to recognize the issues limiting the adoption of program understanding technology.

Tools have their own implied methodologies and processes. These implied processes need to coexist within an analysis process that is naturally goal-directed. However, some tools have substantial preparation and forward-feeding activities. This and other methodology overhead may account partly for the difficulty of introducing certain program understanding tools into the development environment, where the tool users have expectations of quick, guaranteed results, with no learning curve.

Lightweight tools that are specialized or adaptable to do a few things very well may be needed for easier technology insertion. Toward this need, Rigi supports tool evolution, where the tool becomes more domain-specific. Strategies such as conceptual modeling and script writing are used to help capture analysis experience and knowledge about the business.

Acknowledgments

Many thanks go to Paul Sorenson for his helpful comments on the paper, Hausi Müller for his support, and Nazim Madhavji and Jacob Slonim for their thoughts on technology transfer.

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 IRIS Federal Centres of Excellence, and the University of Victoria.

References

[1] Robert B. Grady. *Practical Software Metrics for Project Management and Process Improvement*. Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[2] Erich Buss and John Henshaw. Experiences in program understanding. Technical Report TR-74-105, IBM Canada Laboratory, Toronto, Ontario, Canada, July 1992.

[3] Kenny Wong, Scott R. Tilley, Hausi A. Müller, and Margaret-Anne D. Storey. Structural redocumentation: A case study. *IEEE Software*, pages 46–54, January 1995.

[4] James H. Cross II, Elliot J. Chikofsky, and Charles H. May Jr. Reverse engineering. *Advances in Computers*, 35:199–254, 1992.

[5] James R. Lyle and Mark Weiser. Automatic program bug location by program slicing. In *Proceedings of the Second International Conference on Computers and Applications*, pages 877–, Beijing, China, June 1987. IEEE Computer Society Press.

[6] Kostas Kontogiannis, Morris Bernstein, Ettore Merlo, and Renato De Mori. The development of a partial design recovery environment for legacy systems. In Gawman et al. [20], pages 206–216.

[7] Victor Basili and Scott Green. Software process evolution at the SEL. *IEEE Software*, pages 58–66, July 1994.

[8] J. Howard Johnson. Identifying redundancy in source code using fingerprints. In Gawman et al. [20], pages 171–183.

[9] Reidar Conradi, Christer Fernstrom, and Alfonso Fuggetta. A conceptual framework for evolving software process. *ACM SIGSOFT Software Engineering Notes*, 18(4):26–35, October 1993.

[10] Tilmann Bruckhaus. The impact of inserting a tool into a software process. In Gawman et al. [20], pages 250–264.

[11] Scott R. Tilley. *Domain-Retargetable Reverse Engineering*. PhD thesis, University of Victoria, Victoria, British Columbia, Canada, 1995.

[12] Kenny Wong. Understanding software architecture and behavior through integrated structural and runtime analysis. Research proposal, 1994.

[13] Hausi A. Müller and Brian D. Corrie. Measuring the quality of subsystem structures. Technical Report DCS-193-IR, Department of Computer Science, University of Victoria, Victoria, British Columbia, Canada, November 1991.

[14] Scott R. Tilley, Hausi A. Müller, and Mehmet A. Orgun. Documenting software systems with views. In *SIGDOC '92: Proceedings of the 10th Annual International Conference on Systems Documentation*, pages 211–219, Ottawa, Ontario, Canada, October 1992. ACM Press.

[15] Scott R. Tilley, Kenny Wong, Margaret-Anne D. Storey, and Hausi A. Müller. Programmable reverse engineering. *International Journal of Software Engineering and Knowledge Engineering*, 4(4):501–520, December 1994.

[16] Hausi A. Müller, Mehmet A. Orgun, Scott R. Tilley, and James S. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5:181–204, 1993.

[17] Kenny Wong. Managing views in a program understanding tool. In Gawman et al. [20], pages 244–249.

[18] Scott R. Tilley. Domain-retargetable reverse engineering II: Personalized user interfaces. In Hausi A. Müller and Mari Georges, editors, *Proceedings of the International Conference on Software Maintenance*, pages 336–342, Victoria, British Columbia, Canada, September 1994. IEEE Computer Society Press.

[19] John Mylopoulos. Conceptual modelling and Telos. Technical Report DKBS-TR-91-3, University of Toronto, Toronto, Ontario, Canada, November 1991.

[20] Ann Gawman, M. Gentleman, Evelyn Kidd, Per-Ake Larson, and Jacob Slonim, editors. *Proceedings of the 1993 CAS Conference*, Toronto, Ontario, Canada, October 1993. IBM Centre for Advanced Studies.