

A Framework for Classifying and Comparing Software Reverse Engineering and Design Recovery Techniques*

Gerald C. Gannod[‡]
Computer Science and Engineering
Arizona State University
Box 875406
Tempe, AZ 85287-5406
E-mail: gannod@asu.edu

Betty H. C. Cheng
Computer Science and Engineering
Michigan State University
3115 Engineering Building
East Lansing, MI 48824-1226
E-mail: chengb@cse.msu.edu

Abstract

Several techniques have been suggested for supporting reverse engineering and design recovery activities. While many of these techniques have been cataloged in various collections and surveys, the evaluation of the corresponding support tools has focused primarily on their usability and supported source languages, mostly ignoring evaluation of the appropriateness of the by-products of a tool for facilitating particular types of maintenance tasks. In this paper, we describe criteria that can be used to evaluate tool by-products based on semantic quality, where the semantic quality measures the ability of a by-product to convey certain behavioral information. We use these criteria to review, compare, and contrast several representative tools and approaches.

1 Introduction

Software maintenance has long been recognized as one of the most costly phases in software development [1]. A software system is termed a *legacy system* if that system has a long maintenance history. Many techniques have been suggested for the maintenance of legacy software as is clearly indicated by the number of surveys that have been used to catalog these techniques [2, 3, 4]. Due to the increasing visibility of the *Year 2000 Problem* (i.e., Y2K) ¹ many more tools have been suggested and subsequently cataloged [5].

*This work supported in part by the National Science Foundation grants CDA-9617310, CCR-9633391, CCR-9407318, CCR-9209873, and CDA-9312389, and NASA Training grant NGT-70376.

[†]This author was supported in part by a NASA Graduate Student Researchers Program Fellowship. A portion of this research was performed while this author was at the NASA Jet Propulsion Laboratory.

[‡]Contact Author.

¹The Y2K problem refers to the potential failure of systems due to the use of a two-digit encoding for the year field in software systems.

Given the large number of tools, identifying an appropriate one for the goals of an individual organization can be difficult. Currently, the information gathered on software maintenance tools focuses on high-level characteristics. That is, the gathered information typically lists the languages that are supported and the type of by-products ² (i.e., artifacts) generated from analyzing the input software with the particular tool. For instance, Bellay and Gall [4] describe capabilities related to usability, parsing speed, type of by-product, editing facilities, and report generation. Based on feedback and interaction with industry, it is our claim that in addition to the information contained in these surveys, it is also useful to perform an evaluation of the actual by-products (e.g., function reports, call graphs, data flow diagrams) in order to gain an understanding of the value of the by-products and how they can facilitate software maintenance activities.

In this paper, we describe a framework for analyzing software reverse engineering and design recovery tools and techniques. Within this framework we provide a context by which software reverse engineering and design recovery tools can be classified according to the underlying approach used to analyze software. In addition, we define several criteria for comparing and contrasting tools according to the *semantic quality* of their by-products, where the semantic quality measures the ability of a by-product to convey certain behavioral information. The remainder of this paper is organized as follows. Section 2, describes background information on technology evaluation and summarizes the contents of previous software maintenance tool surveys. A taxonomy is defined in Section 3 as a means for comparing and contrasting different reverse engineering and design re-

²We use the term *by-product* to refer to any artifact that is generated by the process of using a tool. In contrast, a *product* would refer to the result of the tool usage context; an understanding of the program.

covery techniques. Section 4 presents several dimensions of analysis for evaluating the semantic quality of the by-products of reverse engineering and design recovery tools. Section 5 compares several tools, and Section 6 draws conclusions and suggests further investigations.

2 Background

Reverse engineering is defined as the analysis of software components and their interrelationships in order to obtain a description of the software at a high level of abstraction [6]. This term is contrasted with reengineering, which is the process of examination, understanding, and alteration of a system with the intent of implementing the system in a new form [6]. Software reengineering is considered to be a better solution for handling legacy code as opposed to developing software from the original requirements. Since the functionality of the existing software has been achieved over a period of time, it must be preserved for many reasons, including providing continuity to current users of the software.

In the context of software maintenance, we define a *structural* abstraction to be a description of a software system that is based on the syntactic properties of a programming language. For example, encapsulation of a sequence of programming statements into a module is a structural abstraction. In contrast, a functional abstraction is a description of a software system that is based on the semantics of a program. That is, a functional abstraction describes program behavior. For instance, if a sequence of statements is grouped into a module, then the high-level description of the function of that module is a functional abstraction. Recent work in reverse engineering has focused on both the derivation of structural and functional abstractions from program code.

2.1 Evaluation of Software Technology

Brown and Wallnau [7] describe a framework for evaluating software technology that is based on two primary goals: (1) understanding how the evaluated technology differs from other technologies, and (2) understanding how these differences address the needs of specific usage contexts. In order to achieve these goals, Brown and Wallnau suggest a three-phase process for technology evaluation. These phases are: 1) Descriptive modeling, 2) Experiment design, and 3) Experiment evaluation. The *descriptive modeling* phase is used to create a context for candidate technologies. A descriptive model is a description of the assumptions concerning features and their relationship to usage contexts [7]. Two types of descriptive models are the *technology genealogy* and the *problem domain habitat*. The technology genealogy describes the historical context for a given technology, and a problem habitat describes how

the features of a given technology can be adopted as well as the benefits of their use.

The *experiment design* phase involves three primary activities: (1) comparative feature analysis, (2) hypothesis formulation, and (3) experiment design. In this phase, the goals are to develop a set of hypotheses about the added value of a technology that can be established by experiments, and to identify the experiments that are used to substantiate or refute the hypotheses [7].

The final phase, *experiment evaluation*, involves performing experiments to confirm or refute the hypotheses. Brown and Wallnau identify a few different classes of experiments that can be useful in evaluating hypotheses [7]. These experiment categories include:

- *Model problems*: narrowly defined problems that are easily addressed by the candidate technologies. Model problems allow alternative technologies to be directly compared.
- *Compatibility studies*: experiments that study how well candidate technologies operate when combined.
- *Demonstrator studies*: full-scale trial applications of a technology.
- *Synthetic benchmarks*: standard contrived problems that can be used to evaluate the differences between candidate technologies.

In this paper, we analyze several reverse engineering support tools using an assessment technique that is similar to the Brown and Wallnau “Technology Delta Framework”. Specifically, we present the results of the descriptive modeling phase, where we describe a hierarchical genealogy of reverse engineering techniques. In addition, we define several semantic dimensions that are used to qualitatively evaluate some representative reverse engineering support tools, an activity that corresponds to constructing a reference model in the experiment design phase in the technology delta framework. Finally, we provide a comparative analysis of several support tools.

2.2 Previous Surveys

The Air Force Software Technology Support Center (STSC) published a two volume report that compiles information about hundreds of tools that are available for reengineering purposes [3]. While the report lists many tools, the descriptions of the tools are often limited to high-level properties, such as supported languages and vendor contact information. Similar surveys by Zvegintzov [2, 5] also collect descriptions of Reengineering and Y2K tools that list properties similar to the ones listed in the STSC report. In addition, the Y2K survey [5] classifies the tools based on their intended capabilities. For instance, some of the categories

used to group tools are based on whether the tools support activities such as *inventory analysis* (e.g., identification of the executable software inventory), *recovering source from object* (e.g., analysis of binaries in the case that source is not available), and *conversion* (e.g., identification of code and data structures that require modification).

A recent survey by Bellay and Gall [4] compares four reverse engineering tools using several criteria that can be used to analyze the effectiveness of the input parsers, analyze the editing and browsing capabilities of the tools, and evaluate the general usability of the tools. While the Bellay and Gall survey provides a more in-depth view of tools when compared to the previous surveys, it focuses primarily on tool properties as opposed to the characteristics and qualities of the tool by-products.

Our approach to classifying and analyzing software reverse engineering and design recovery tools and techniques is intended to provide a framework for assessing the quality and usability of the by-products. As such, this paper provides a complementary approach to the assessment and comparison of tools such as those contained in the surveys described above.

2.3 Tool Capabilities

Von Mayrhauser and Vans [8] observed that maintenance engineers use an integrated set of mental models during design recovery. Based on this observation, a matrix was developed that maps different tool capabilities to activities associated to the different components of the integrated set of mental models.

Storey *et al.* [9] identified a number of cognitive design elements that can be used to form a mental model of a software system during program comprehension activities. These design elements can be used to identify a set of quality attributes that reverse engineering and design recovery tools should implement in order to facilitate certain activities such as *navigation*, *top-down comprehension*, *bottom-up comprehension*, and *reduction of disorientation*.

The approach described in this paper provides a set of criteria for evaluating reverse engineering and design recovery that are complementary to those identified by Von Mayrhauser and Vans [8] and Storey *et al.* [9]. Accordingly, the approach described in this paper has the potential of providing complementary assistance to both tool adopters and tool developers.

3 Classification

In order to classify automated and semi-automated reverse engineering techniques, we have developed the hierarchical taxonomy shown in Figure 1. At the highest level, the techniques can be subdivided into two classes: *informal* and *formal*. This particular decomposition was chosen because it depicts the high-level methodology of tech-

niques, as opposed to the mental model approach used by Von Mayrhauser and Vans [8], and Storey *et al.* [9]. *Informal* approaches are those methods that rely on pattern matching and user-driven clustering techniques based on the syntactic structure of code. The informal techniques facilitate the derivation of structural and functional abstractions. The pattern matching and clustering techniques are considered informal because the design representations that are constructed are informal and the techniques lack a rigorous method for verifying consistency between source and design. The *formal* approaches are those techniques that are based on using some type of formal analytical method for deriving a specification from source code. The basis for the formal techniques are grounded in mathematical logic so that each step can be formally verified. The primary difference between the informal techniques and the formal techniques is the use of formal specification languages that have well-defined syntax and semantics. In addition, the formal techniques have associated inference rules that can be used to construct proofs in order to rigorously verify the correctness of each step of the reverse engineering process. As such, the formal techniques facilitate the derivation of functional abstractions from program code.

Various reverse engineering and program understanding techniques can be evaluated and classified using the taxonomy shown in Figure 1. The utility of classifying tools using this taxonomy is that it provides a means for determining the current trends in supporting reverse engineering and design recovery, and aids in identifying the areas that require further investigation. As a notational convention, an acronym follows the name of each approach to indicate the classification of the technique within the taxonomy. For instance, a tool “**foo**” might fall within class “**IPLC**” to indicate that the technique is an informal, plan-based, commercial tool. The annotations at the leaves of the classification hierarchy in Figure 1 associate each tag to a location in the classification.

3.1 Informal Techniques

In the context of reverse engineering and program understanding, a technique is classified as *informal* if the methods used to recover designs from source code is based on pattern matching or analysis of syntactic structures as opposed to semantic structures. The informal techniques can be decomposed into two additional sub-categories: *plan-based* and *parsing-based*. The *plan-based* techniques rely primarily on using pattern matching to identify clichés or *plans* within source code and have been a major focus in both research and commercial organizations. A program plan is a description of a computational unit contained within a program where a computational unit performs some abstract function [10]. A program plan can be *localized* or *de-localized* in the sense that the code recognized as satis-

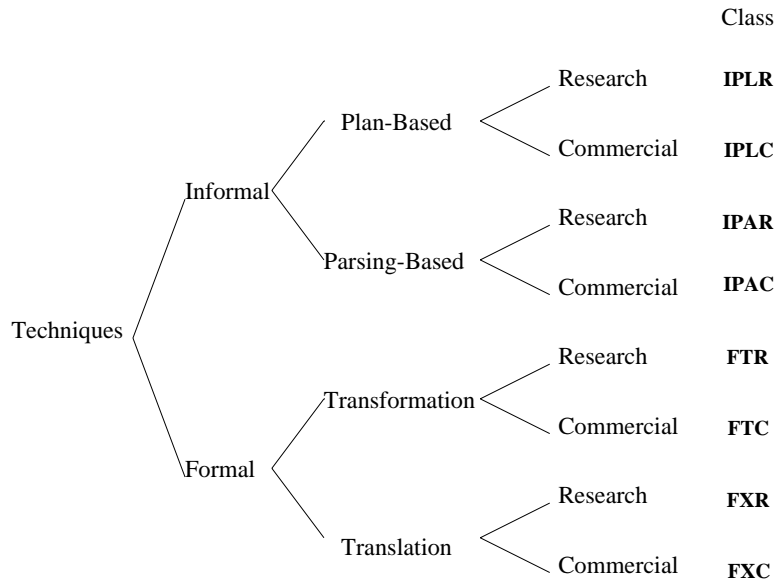


Figure 1. A Taxonomy of Reverse Engineering Techniques

ying the plan can be located in contiguous (localized) or non-contiguous (de-localized) sequences of code [11]. To date, most plan-based approaches have been developed by research organizations [12, 13, 14], although some industrial adoption of this approach is occurring [15].

A *parsing-based* approach is one in which a program is analyzed using the properties of the syntactic structure of a programming language. In general, the parsing-based approach is used to construct a high-level structural abstraction of the source code. These abstractions typically come in the form of data flow diagrams or some other graphical representation of the design. A significant number of commercial tools use a parsing-based technique for supporting reverse engineering [16, 17], and research organizations continue to investigate the use of advanced parsing-based approaches [18, 19].

3.2 Formal Techniques

Formal methods for software development are analytical techniques for assuring, by construction, that a derived specification is correct with respect to some other specification. A reverse engineering technique is formal if the steps of the method have a formal mathematical basis. When applied to reverse engineering, a formal method takes as input a source program (e.g., a low-level specification) and derives a formal specification. In the formal context, reverse engineering techniques can be subdivided into two categories: techniques that use a knowledge-base or *transformation* library to derive formal specifications from code,

and techniques that use derivation or *translation* to derive formal specifications from code.

A *transformation* is a means for changing a specification from one form to another while preserving the semantics of the specification. In the context of programs, a *program transformation* is a means for changing a program from one form to another while preserving the semantics of the program. Each program transformation is typically used to change a group of programming statements at a time, where the group is determined by the author of the particular transformation.

Transformation is contrasted with *translation*, where a translation is also a means for changing a program from one form to another while preserving semantics but at an atomic level of granularity. The primary difference between transformation and translation is the degree to which high-level knowledge about a problem domain or programming language is incorporated into the transformation or translation rules. In the case of transformation, the rules typically involve transforming aggregations of programming statements into simpler, equivalent sequences of statements (as is the case in restructuring transformations) or concise formal specifications. In many cases, a large library of transformations is required to capture the many different possible code constructions. Translation, in contrast, involves much simpler rules that are based on single atomic statements such as assignments, conditionals, and iteratives, thus requiring fewer rules. A program compiler can be considered a translator since each program statement is translated into an equivalent binary form. In the context of

program reverse engineering, a translation technique is one that translates a program into an equivalent formal specification.

Research into the use of formal methods for reverse engineering has addressed both the use of transformation [20, 21] and translation [22]. Industrial adoption of formal techniques has begun but is limited [23, 24].

4 Semantic Dimensions

A *by-product* is an artifact that is constructed by a reverse engineering tool as a result of analyzing program code. One way to evaluate the by-products of a tool or technique is to simply list the formats and representations that are produced by a particular tool. For instance, one tool might produce reports about the formats of data structures as well as visual representations such as call graphs and data-flow diagrams. While this knowledge about a tool is extremely helpful, it is of equal importance to understand the nature of these by-products and to evaluate a tool based on this information. In order to analyze the value of the by-products of the various tools, we define four semantic dimensions: *distance*, *accuracy*, *precision*, and *traceability*. These measures enable a software maintainer to evaluate a tool based on the level of importance placed on the consistency between an abstract representation as compared to a given implementation.

4.1 Semantic Distance

The *semantic distance* describes the number of levels of abstraction that separate the input and output of a particular technique. The semantic distance is a relative distance, since no absolute measure of abstractness can reasonably be developed. Instead, a subjective measure based on the level of algorithmic detail must be considered.

As a rule of thumb, the greater the semantic distance, the more abstract the by-product. Suppose, for instance, we translated source code from FORTRAN to C. Since there is no difference in the level of abstraction between the two representations, the semantic distance is small or non-existent. On the other hand, if we reverse engineer source code from C into a data-flow diagram representation, the semantic distance is greater. At the extreme, we might reverse engineer source code from C into a description of the concept of the program; a transformation that would result in the greatest semantic distance.

A concept related to the semantic distance is the *inter-step distance* that measures the semantic distance between each intermediate step of a technique. For example, if a reverse engineering technique consists of three steps, where each step produces a representation that is more abstract than the previous step, the semantic distance that separates each step in the technique is the inter-step distance.

Figure 2 summarizes the semantic distance. The left-hand side of the diagram shows the different abstraction targets for deriving by-products from source code. The right-hand side of the diagram describes the relative distance between each of the targeted levels.

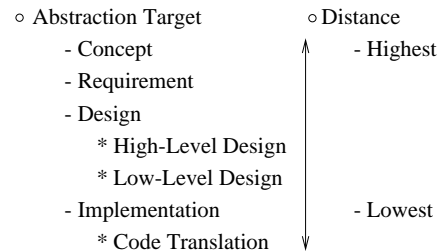


Figure 2. Semantic Distance

4.2 Semantic Accuracy

The *semantic accuracy* describes the level of confidence that a specification is correct with respect to the input source code. Many of the by-products derived from an analysis of syntactic information rarely have low semantic accuracy. That is, the information that is recovered from the source code is accurate with a high degree of confidence. In contrast, the techniques that derive by-products based on semantic information may not be as accurate. For instance, those techniques that are based on the plan abstraction approach may rely on the assumption that plans are not interleaved [11], and, as such, may ignore the effect of *cancellation* or composition in their description of a particular sequence of software. That is, two or more program plans may be identified in the same sequence of code, but their combined effects may not be well-understood and thus, the accuracy of the design abstraction may be reduced.

Some of the factors that impact the semantic accuracy of a given technique are the number of analysis stages and the inter-step distances between the stages. Since each stage of a recovery technique results in loss of information, the composition of applying each step results in an increased potential for a loss of accuracy.

Figure 3 summarizes semantic accuracy. The left-hand side of the diagram shows the different techniques used to derive by-products from source code. The right-hand side of the diagram describes the relative accuracy of each of the methods.

4.3 Semantic Precision

Semantic precision describes the level of detail of a specification and the degree that the specification is formal. A formal specification is the most precise given the well-defined syntax and semantics associated with this form of

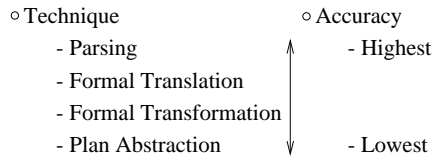


Figure 3. Semantic Accuracy

description. The least precise by-product is natural language due to its potential for ambiguity. A more precise specification is apt to be more amenable to automated analytical processing while a less precise specification is better suited for discussion between developers.

Figure 4 summarizes semantic precision. The left-hand side of the diagram shows the different by-products that can be derived from program code. The right-hand side of the diagram describes the relative precision of each of the by-products.

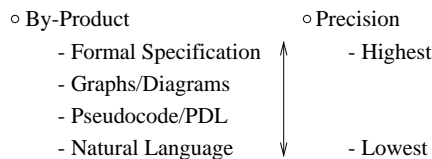


Figure 4. Semantic Precision

4.4 Semantic Traceability

Semantic traceability describes the degree that a specification can be used to reconstruct an equivalent program. Semantic traceability highly depends on the semantic accuracy and semantic precision of the end by-product. That is, accuracy contributes to the degree to which the original program and the new program correspond semantically, and precision contributes to the degree that the representation is free of ambiguity. The semantic precision has an impact on the amount of semantic information that can be used to construct the new program. As such, the by-products that contain functional abstractions facilitate traceability, with formal by-products facilitating the highest degree of traceability. The ability of a programmer to reproduce a working system varies greatly between a formal specification and a graphical design since semantic information is contained in the formal specification while, in general, only syntactic information is contained in a graphical design.

Figure 5 summarizes semantic traceability. The left-hand side of the diagram shows the different by-products that can be derived from program code. The right-hand side

of the diagram describes the relative traceability of each of the by-products.

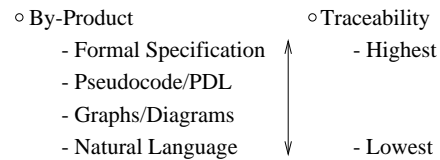


Figure 5. Semantic Traceability

4.5 Discussion

Ideally, a design derived from program code has a balance between all of the semantic dimensions. A large semantic distance may produce a more abstract specification but if that specification lacks accuracy and precision, there is a low degree of confidence that the specification captures the actual functionality of the source code. On the other hand, a specification with a high degree of precision and traceability that lacks a reasonably large semantic distance may have a level of detail that does not facilitate program understanding. In the end, it is the software maintenance programmer that must weigh the goals of a project against the relative advantages and disadvantages offered by the by-products of the various techniques in order to make the appropriate decision for a particular project or organization.

The primary utility of the semantic dimensions described in this section is to facilitate the classification of tools based on the technique used by a tool to construct a by-product, as well as the quality of the by-product. Several guidelines can be used to provide a context to a tool evaluator in order to select tools appropriate for the specific maintenance task to be performed. Table 1 suggests the appropriateness of applying tools with by-products that have specific characteristics for certain software maintenance activities. In the table, cells are marked with the letters “H” and “L” to indicate, respectively, the high and low end of each of the ranges presented in Figures 2 through 5. In addition, some of the possible assignments, such as LLLL (corresponding to low distance, accuracy, precision, and traceability), have no application as indicated by the “NA”. Other assignments correspond directly to tools described later in this paper. Finally, some of the assignments correspond to activities that are not yet fully realized, such as full-scale reengineering.

5 Comparison

In this section we evaluate different reverse engineering and design recovery approaches by comparing them based on surface or *informational* criteria as well as the semantic dimensions of the tool by-products. The tools compared in

<i>Goals - Objectives or Applications</i>	<i>Dimension</i>			
	Distance	Accuracy	Precision	Traceability
Reengineering	H	H	H	H
Simulation	H	H	H	L
Functional Abstraction	H	H	L	H
Structural Abstraction	H	H	L	L
Functional Analysis	H	L	H	H
NA	H	L	H	L
Requirement/Concept recovery	H	L	L	H
NA	H	L	L	L
Debugging, Re-code, Language Conversion	L	H	H	H
Structural Analysis	L	H	H	L
Algorithm Understanding	L	H	L	H
Structure Understanding	L	H	L	L
NA	L	L	H	H
NA	L	L	H	L
NA	L	L	L	H
NA	L	L	L	L

Table 1. Suggested Applications Classified by Dimension

this section were selected based on a number of criteria including availability of the tools and availability of papers or reports describing the technology. Due to space constraints, detailed descriptions of the individual tools are not included, but may be found elsewhere [25].

5.1 Comparison Criteria

The criteria to be used in comparing the different approaches are subdivided into two groups: *informational* and *evaluational*. The informational criteria are a high-level list of surface characteristics, such as source language, platform, and technique. These criteria serve to provide a quick glance index to the reader and a means for quickly finding more information about a system if so desired. The evaluational criteria are a list of detailed characteristics that allow a user to evaluate the differences between the respective approaches. These characteristics include extensibility, high-level abstractions, formal specifications, metrics, standard diagrams, precision level, and traceability level.

5.1.1 Informational Criteria

The informational comparison provides a quantitative means for measuring each of the tools described in this paper. That is, each of the criteria can be used as a feature “checkbox” for a tool. In this paper, we use a small set of informational criteria consisting of *Languages*, *Platforms*, and *Techniques*. Bellay and Gall [4] list several other criteria of this type. The language criteria indicate the languages supported by a particular tool. The languages that the various tools support include *C*, *C++*, *COBOL*, *ADA*, and *FORTRAN*. Platform criteria are used to indicate on

which hardware platforms the tools can execute. The platforms include support for *PC*, *Sun*, *IBM RS6000*, *HP*, and *Macintosh* although we focus only on PC and Sun support. The approach (e.g., informal/formal, plan/parsing, etc.) is also used to further classify each technique. The survey by Bellay and Gall covers many more characteristics that are informational in nature [4]. Our intention here is to provide a quick coverage of basic features that can be combined with the criteria described in Section 5.1.2 to provide a general overview of tool capabilities and the semantic quality of the tool by-products.

5.1.2 Evaluational Criteria

Evaluational criteria provide a more in-depth means for categorizing different tools by differentiating tools according to by-products. In this paper, by-products include structure charts, flow diagrams, data dictionaries, metrics, complexity measures, and formal specifications. Another type of evaluational criteria is the *Open Interface* characteristic that indicates whether a tool has an application programming interface (API) to allow users to build applications. In addition, this paper compares the characteristics of the by-products by indicating whether the tools produce structural or functional abstractions. Within this categorization, the abstractions are classified as either *as-built* (e.g., low-level) or *abstraction* (e.g., high-level). Finally, the evaluational comparison describes the by-products using the four semantic dimensions described in Section 4 (i.e., distance, accuracy, precision, and traceability) in the range of low (L), medium-low (M/L) medium-high (M/H) and high (H).

5.1.3 A note about by-products

Tool *by-products* are the artifacts generated by tools as a result of program analysis. Using the informational and evaluational criteria, different inferences can be made about the value of a tool with respect to the by-products. For instance, a formal specification is a form of by-product that has the properties of being precise, and in general, traceable. However, formal specifications are not generally perceived to be user-friendly (that is, they may require some specific background education). Additionally, structure charts are user-friendly, precise, and traceable but lack high-level abstraction. In the remainder of this section we evaluate the primary by-products of each tool. We also provide an evaluation of the by-products along each of the semantic dimensions described in Section 4. Inferences about usability, productivity, etc. are all dependent on the final end users.

5.2 Informational Comparison

In this section we compare a number of tools based on the informational criteria listed in Section 5.1.1. The group of tools evaluated here is by no means comprehensive. Rather, the tools presented here are a representative set of available techniques that are being used to demonstrate the evaluation method described in this paper. In addition, the by-products identified in this analysis is not intended to be comprehensive, but instead describes a sample of the possible representations. An index of tools is provided in Tables 2 and 3. Tables 4 and 5 summarize the tools based on the informational criteria.

SR	=	Software Refinery [26]
VR	=	McCabe VRT [17]
4D	=	Imagix 4D [16]
XI	=	Xinotech Research [15]
LS	=	Logiscope [27]
EN	=	Ensemble Software [28]
DM	=	Design Maintenance System [23]

Table 2. Index of Commercial Tools

Table 4 compares commercially available tools using the informational criteria. This table shows that C and COBOL are the most widely supported languages among commercial tools and that the McCabe VRT tool supports the greatest number of languages. Among platforms, Sun is the most widely supported, although in this comparison we make no distinction between the Solaris and SunOS Operating Systems. Again, the McCabe VRT tool supports the most platforms. Among the techniques used, parsing-based is the most popular. Of note is the fact that the Software Refinery supports the use of transformations although the built-in tools do not use formal transformation as an analysis

PA	=	PAT [29]
CS	=	COBOL/SRE [13, 30]
DE	=	DECODE [12]
LT	=	LANTRN [14]
MA	=	Maintainer's Assistant [20, 31, 32, 33]
RE	=	REDO Toolset [21]
RI	=	Rigi [18, 34]
AS	=	AutoSpec [22, 35]
RM	=	RMTTool [19, 36]

Table 3. Index of Research Tools

technique. Finally, of all the commercial tools, only the Xinotech tool uses a plan-based approach.

		SR	VR	4D	XI	LS	EN	DM
Languages	C	•	•	•	•	•		
	C++		•	•		•		•
	COBOL	•	•		•	•	•	
	ADA	•	•		•	•		
	FORTTRAN	•	•		•	•		
Other		•		•	•			
Platform	PC		•				•	•
	Sun	•	•	•	•	•	•	
Technique	Plan-Based				•			
	Parsing-Based	•	•	•		•	•	
	Transformation	•						•
	Translation							

Table 4. Comparison of Commercial Tools by informational criterion

Table 5 compares research tools using the informational criteria. This table shows that, like the commercial tools, C and COBOL are the most widely supported languages. Of the research tools, Rigi supports the most languages (COBOL, C, C++). "Other" languages are also more widely supported than FORTRAN and ADA due to the fact that most research tools use source languages that resemble production languages with the caveat that translation to and from production languages from the research languages is theoretically possible. Among platforms, Sun is supported most often, with the Rigi system supporting the most platforms (Unix, Windows). The approaches used by the research tools are divided mainly into two groups: those approaches that use plan-based techniques, and those approaches that use some formal technique. Only the Rigi system and RMTTool use a parsing-based technique.

	PA	CS	DE	LT	MA	RE	RI	AS	RM
Languages	C			•			•	•	•
	C++						•		
	COBOL		•	•		•	•		
	ADA				•				
	FORTRAN								
	Other	•			•	•			•
Platform	PC						•		•
	Sun		•	•		•	•	•	•
Technique	Plan-Based	•	•	•	•				
	Parsing-Based						•		•
	Transformation					•	•		
	Translation							•	

Table 5. Comparison of Research Tools by informational criterion

Parsing-based techniques are the most widely used technique among the commercial tools, which reflects the fact that the parsing techniques are more mature. The research tools focus on the use of plans or formal methods, although the plan-based technique has been adopted by the commercial tool offered by Xinotech. A possible conjecture is that the plan-based approach is becoming more mature and is beginning to be adopted by industry, although such a conjecture should be taken lightly since our sample size is relatively small.

5.3 Evaluational Comparison

Evaluational criteria provide a more qualitative means for comparing the various approaches. Tables 6 and 7 summarize the by-products produced by each tool, grouped by commercial tools and research tools, respectively. Tables 8 and 9 summarize the characteristics of the by-products using the criteria described in Section 5.1.2. Again, these tables are grouped by commercial and research tools, respectively.

Table 6 shows the by-products of the various commercial tools. Among commercial tools, creation of structure charts is the most widely supported activity and the McCabe VRT and the Ensemble tools create the largest number of by-products. The Software Refinery and Xinotech tools provide support for user-defined applications via their programmer interfaces. Of all the commercial tools, none support the construction of formal specifications, and only the Xinotech tool creates functional abstractions in the form of recognized program plans.

	SR	VR	4D	XI	LS	EN	DM
Structure Charts	•	•	•		•	•	
Flow Diagrams		•			•	•	
Data Dictionaries	•	•	•			•	
Metrics		•			•	•	
Complexity Measures		•			•	•	
Formal Specifications							
Other		•	•	•	•		•
Open Interface	•			•			

Table 6. Comparison of Commercial Tools by By-products

	PA	CS	DE	LT	MA	RE	RI	AS	RM
Structure Charts							•	•	•
Flow Diagrams							•		•
Data Dictionaries									
Metrics					•		•		
Complexity Measures									
Formal Specifications				•	•	•		•	
Other	•	•	•						
Open Interface							•		

Table 7. Comparison of Research Tools by By-products

Table 7 shows the by-products of the various research tools. Most research approaches focus on the creation of either formal specifications or some other functional abstraction with only the Rigi and Reflexion Model tools supporting the creation of structural by-products and abstractions.

Overall, the main difference between the commercial and the research tools is the nature of the by-products. That is, the research by-products focus on creating functional abstractions whereas the commercial by-products focus on generating structural abstractions, as shown in Tables 8 and 9. Specifically, Table 8 shows that only the Xinotech tool produces functional abstractions while Table 9 shows that only the Rigi and Reflexion Model tools produce structural abstractions. Tables 8 and 9 also show the difference between commercial and research tools with respect to the semantic dimensions (i.e., distance, accuracy, precision, and traceability) of the by-products. The commercial

by-products tend to have a low semantic distance but are accurate in their representations. On the other hand, the research tools have a high degree of semantic distance but the accuracy tends to suffer. A few of the research tools also focus on higher precision but few do well in terms of traceability and accuracy.

		SR	VR	4D	XI	LS	EN	DM
Structural	As-built	•	•	•		•	•	
	Abstraction							
Functional	As-built							
	Abstraction				•			•
Dimensions	Distance	L	L	L	H	L	L	L
	Accuracy	H	H	H	M	H	H	H
	Precision	L	L	L	L	L	L	H
	Traceability	L	L	L	M/L	L	L	H

Table 8. Comparison of Commercial Tools by evaluational criterion

		PA	CS	DE	LT	MA	RE	RI	AS	RM
Structural	As-built							•	•	•
	Abstraction							•		•
Functional	As-built									
	Abstraction	•	•	•	•	•	•		•	
Dimensions	Distance	H	H	H	M	M	M	H	L	H
	Accuracy	M	M	M	M	M	M	M	H	H
	Precision	L	L	L	H	H	H	L	H	L
	Traceability	M/L	M/L	M/L	M/L	M/H	M/H	L	H	L

Table 9. Comparison of Research Tools by evaluational criterion

5.3.1 Semantic Distance

The semantic distance measures the gap between the input source code and a tool by-product. The semantic distance of the by-products derived by many of the commercial tools can typically be classified as falling within the low to medium range. By comparison, the by-products of many of the research tools fall within the medium to high range. The explanation for these observations can be traced back to the type of by-products being constructed and the techniques used to generate them. The commercial tools have

traditionally focused on using parsing techniques to derive structural abstractions, while in comparison, the research tools have focused on the use of many different techniques to derive functional abstractions.

5.3.2 Semantic Accuracy

Semantic accuracy provides a measure of the correctness, either behavioral or structural, of a by-product when compared with the original source code. The semantic accuracy of many of the commercial tools can be classified as falling within the medium to high range. The research tools tend to be in the medium-low to medium-high range. The exceptions to this characterization are the DMS and Xinotech systems which, incidently, incorporate many of the techniques typically used by the research organizations. The AUTOSPEC tool has an accuracy that is in the medium to high range at the cost of semantic distance between source code and abstraction target. Finally, the RM-Tool has a higher level of accuracy but at the cost of requiring a high degree of user intervention to introduce abstractions. The observation that the commercial tools are more accurate than the research tools can be explained, again, by the nature of the techniques used to derive the by-products. The commercial tools rely primarily on parsing techniques to derive structural abstractions and do not focus on behavioral or semantic concerns. As such, the tools benefit from focusing on information that can be easily derived using syntactic constructs of the input languages. Since the research tools are focusing on the much harder problem of deriving function or behavior, there is a certain degree of information loss that has an effect on the accuracy of the by-products.

5.3.3 Semantic Precision

Semantic precision measures the degree of formality in the representations of recovered designs. In the commercial tools, the degree of formality tends to lie in the low to medium-low range. This trend is explained by the graphical nature of the by-products. The degree of formality in the by-products produced by the research tools varies greatly as some fall in the low to medium-low range, while others fall into the medium to high range. The large variance between the research by-products reflects the different philosophies behind the techniques used to derive the abstractions. Those by-products with lower precision tend to use informal techniques while the higher precision representations are by-products of more formal approaches. The LANTeRN tool is a notable exception to these trends in that the approach is based on an informal technique while the by-product used a formal notation.

5.3.4 Semantic Traceability

Semantic traceability measures the degree that a by-product can be used to facilitate code reconstruction. The

commercial tools largely fall on the lower end of the traceability scale since the by-products largely describe structure but rarely describe behavior. The research tools vary in this dimension from low to high. The techniques that utilize the plan-based approach fall into the medium range since the behavior that is captured by the by-products can be used as a starting point for redevelopment. The tools that utilize a formal approach fall into the medium to high range. The by-products (i.e., specifications) that are produced by these approaches capture information that can be used in formal code synthesis via transformations as well as for formal verification steps during specification refinement. In contrast to many of the commercial tools, the DMS tool can support this same class of activity as well.

6 Conclusions

Many tools that support software reverse engineering and design recovery focus on facilitating the understanding of the structure of an existing software system. Maintenance programmers use this knowledge of the structure along with their own experience to form a model of the functionality of the system. Many tools and techniques exist that support the construction of functional abstractions that provide further information to the maintenance programmer which facilitate the formation of more accurate models.

In this paper we have described several criteria that can be used to compare and contrast different approaches for reverse engineering and design recovery. In contrast to other surveys that focus on tool properties, this paper focuses primarily on the by-products of the tools. We have observed that different approaches can potentially provide complementary information that in the long run, when used together, can improve overall program understanding activities. In order to continue to assess the value of the comparison criteria described in this paper, we intend to develop and perform several experiments that will allow us to study the effect of applying multiple approaches to the analysis of software systems. In addition, we intend to extend the scope of our analysis to include many other tools in order to further analyze and assess the current trends and future directions of reverse engineering and design recovery tool development.

Acknowledgments

The authors wish to thank the anonymous reviewers who provided comments on this paper. The authors also wish to thank Hausi Müller for comments on a previous version of this work.

References

- [1] Roger S. Pressman. *Software Engineering A Practitioner's Approach*. McGraw-Hill, fourth edition, 1997.
- [2] N. Zvegintzov, editor. *Software Management Technology Reference Guide*. Software Management News Inc., 1994.
- [3] M. R. Olsem and C. Sittenauer. Reengineering technology report (vol. 1 and 2). Technical report, Software Technology Support Center, Hill AFB, 1995.
- [4] Berndt Bellay and Harald Gall. A Comparison of four Reverse Engineering Tools. In *Proceedings for the Fourth Working Conference on Reverse Engineering*, pages 2–11. IEEE, 1997.
- [5] N. Zvegintzov. A Resource Guide to Year 2000 Tools. *Computer*, 30(3):58–63, March 1997.
- [6] Elliot J. Chikofsky and James H. Cross. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, January 1990.
- [7] Alan W. Brown and Kurt C. Wallnau. A Framework for Evaluating Software Technology. *Software*, 13(5):39–49, September 1996.
- [8] A. von Mayrhauser and A. M. Vans. From code understanding needs to reverse engineering tool capabilities. In *Proceedings of Computer Aided Software Engineering*, pages 230–239. IEEE, July 1993.
- [9] Margaret-Anne Storey, F.D. Frachia, and Hausi A. Müller. Cognitive design elements to support the construction of a mental model during software visualization. In *Proceedings of the 5th International Workshop on Program Comprehension*, pages 17–28. IEEE, May 1997.
- [10] Charles Rich and Richard C. Waters. *The Programmer's Apprentice*. ACM-Press, 1990.
- [11] Spencer Rugaber, Kurt Stirewalt, and Linda Wills. The Interleaving Problem in Program Understanding. In *Proceedings for the Second Working Conference on Reverse Engineering*. IEEE, 1995.
- [12] Alex Quilici. A Memory-Based Approach to Recognizing Program Plans. *Communications of the ACM*, 37(5):84–93, May 1994.
- [13] Jim Q. Ning, Andre Engberts, and Wojtek Kozaczynski. Automated Support for Legacy Code Understanding. *Communications of the ACM*, 37(5):50–57, May 1994.
- [14] Salwa K. Abd-El-Hafiz and Victor R. Basili. A Knowledge-Based Approach to the Analysis of Loops. *Transactions on Software Engineering*, 22(5):339–360, May 1996.
- [15] Xinotech. [Online] Available <http://www.xinotech.com/tech-overview.html>.
- [16] Imagix 4D. [Online] Available <http://www.teleport.com/~imagix>.
- [17] The McCabe Visual Reengineering Toolset. [Online] Available <http://gate.mccabe.com/visual/reeng.html>.

- [18] Scott R. Tilley, Kenney Wong, Margaret-Anne Storey, and Hausi A. Müller. Programmable Reverse Engineering. *The International Journal of Software Engineering and Knowledge Engineering*, 4(4):501–520, 1994.
- [19] G. C. Murphy, D. Notkin, and K. Sullivan. Software Reflexion Models: Bridging the Gap between Source and High-Level Models. In *Proceedings of the third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1995.
- [20] M. Ward, F.W. Calliss, and M. Munro. The Maintainer's Assistant. In *Proceedings for the Conference on Software Maintenance*. IEEE, 1989.
- [21] J.P. Bowen, P.T. Breuer, and K. Lano. The REDO Project: Final Report. Technical Report PRG-TR-23-91, Oxford University, 1991.
- [22] Gerald C. Gannod and Betty H. C. Cheng. Strongest Post-condition as the Formal Basis for Reverse Engineering. *Journal of Automated Software Engineering*, 3(1/2):139–164, June 1996. A preliminary version appeared in the *Proceedings for the IEEE Second Working Conference on Reverse Engineering*, July 1995.
- [23] Ira D. Baxter and Michael Mehlich. Reverse Engineering is Reverse Forward Engineering. In *Proceedings of the Fourth IEEE Working Conference on Reverse Engineering*. IEEE, October 1997.
- [24] Peritus Software Services. [Online] Available <http://www.peritus.com/>.
- [25] Gerald C. Gannod and Betty H. C. Cheng. On the Classification and Comparison of Software Reverse Engineering and Design Recovery Techniques. Technical Report MSUCPS-TR98-35, Michigan State University, November 1998. Submitted to ICSM'99.
- [26] Lawrence Markosian, Philip Newcomb, Russell Brand, Scott Burson, and Ted Kitzmiller. Using an Enabling Technology to Reengineer Legacy Systems. *Communications of the ACM*, 37(5):58–70, May 1994.
- [27] Verilog logiscope. [Online] Available <http://www.verilogusa.com/log/logiscop.htm>.
- [28] Cayenne ensemble. [Online] Available <http://www.cayennesoft.com/products/datasheets/ensemsoft.html>.
- [29] Mehdi T. Harandi and Jim Q. Ning. Knowledge-Based Program Analysis. *IEEE Software*, 7(1):74–81, January 1990.
- [30] Wojtek Kozaczynski and Jim Q. Ning. Automated Program Understanding by Concept Recognition. *Automated Software Engineering*, 1(1):61–78, 1994.
- [31] Martin Ward. Abstracting a Specification from Code. *Journal of Software Maintenance: Research and Practice*, 5:101–122, 1993.
- [32] Tim Bull. An Introduction to the WSL Program Transformer. In *Proceedings for the Conference on Software Maintenance*, pages 242–250. IEEE, 1990.
- [33] E. Younger, Z. Luo, K. Bennett, and T. Bull. Reverse Engineering Concurrent Programs using Formal Modelling and Analysis. In *Proceedings of the 1996 International Conference on Software Maintenance*, pages 255–264. IEEE, 1996.
- [34] K. Wong, S. R. Tilley, H. A. Müller, and M.-A. D. Storey. Structural redocumentation: A case study. *IEEE Software*, pages 46–54, January 1995.
- [35] Gerald C. Gannod and Betty H. C. Cheng. Using Informal and Formal Methods for the Reverse Engineering of C Programs. In *Proceedings of the 1996 International Conference on Software Maintenance*, pages 265–274. IEEE, 1996. Also appears in the Proceedings for the Third IEEE Working Conference on Reverse Engineering.
- [36] Gail C. Murphy and David Notkin. Reengineering with Reflexion Models: A Case Study. *Computer*, 30(8):29–36, August 1997.