Yugoslav Journal of Operations Research
9 (1999), Number 2, 257-271

A CLASSIFICATION OF REVERSE ENGINEERING TOOLS AND CRITERIA FOR THEIR EVALUATION AND SELECTION

Dragan BOJIĆ, Dušan VELAŠEVIĆ

University of Belgrade, Faculty of Electrical Engineering
Bulevar Revolucije 73, 11120 Belgrade, Yugoslavia
{bojic, velasevic}@buef31.etf.bg.ac.yu

Abstract: In the current trend of evolutionary software life cycles and the reuse of software components, great emphasis is put onto the reverse engineering and reengineering of existing software systems. This paper presents a classification of commercial tools that support reverse engineering and reengineering activities. The tools are classified according to the domains of their application: maintenance, reverse design, redocumentation, metrics analysis, restructuring, objectification, reusable component extraction, etc. Also presented is a general framework for the process of reverse engineering tool evaluation and selection by a potential user with emphasis on the criteria for their quantitative evaluation.

Keywords: Software engineering, reverse engineering, software tools classification, software tool evaluation.

1. INTRODUCTION

In the modern process of software production, the maintenance and functional enhancement of existing systems represent significant and costly activities in their entire life cycle. It is estimated [11] that 30-35% of total life-cycle costs are consumed in trying to understand software after it has been delivered and make changes - representing 60-70% of maintenance costs. The number of complex systems built from scratch is steadily decreasing, while at the same time there are more and more legacy systems in every area of computer usage.

Program understanding is the process of developing mental models of a software system's intended architecture, meaning, and behavior [9]. Program understanding is accomplished by the reverse engineering process, which consists of the following activities:

1. Model: Construct domain-specific models of the application using conceptual modeling techniques.

2. Extract: Gather the raw data from the subject system using the appropriate extraction mechanisms.

3. Abstract: Create abstractions that facilitate program understanding and permit the navigation, analysis, and presentation of the resultant information structures.

Reverse engineering is a constitutive element of software reengineering - the process of examining and altering a subject system to reconstitute it in a new form, to improve one's understanding of software, or to prepare or improve the software itself for increased maintainability, reusability, or evolvability (Fig. 1). The activities and tools described in this paper are conceived to support the process of understanding legacy systems and transforming them into evolutionary systems [5] - systems which are adaptable to changing functional requirements and the implementation of technology throughout their entire life cycle.

2. APPLICATION AREAS

The application areas for reverse engineering and reengineering tools include software maintenance, redocumentation, design recovery, metric analysis, restructuring, source code translation, migration to another hardware platform/operating system, objectification, reusable component extraction, year 2000 problem solving and others [4]. Table 1 presents some basic data about several popular tools classified according to their main functionality, based on [1], WWW and tool documentation. The rest of the section is concerned with a short description of the application areas of these tools.

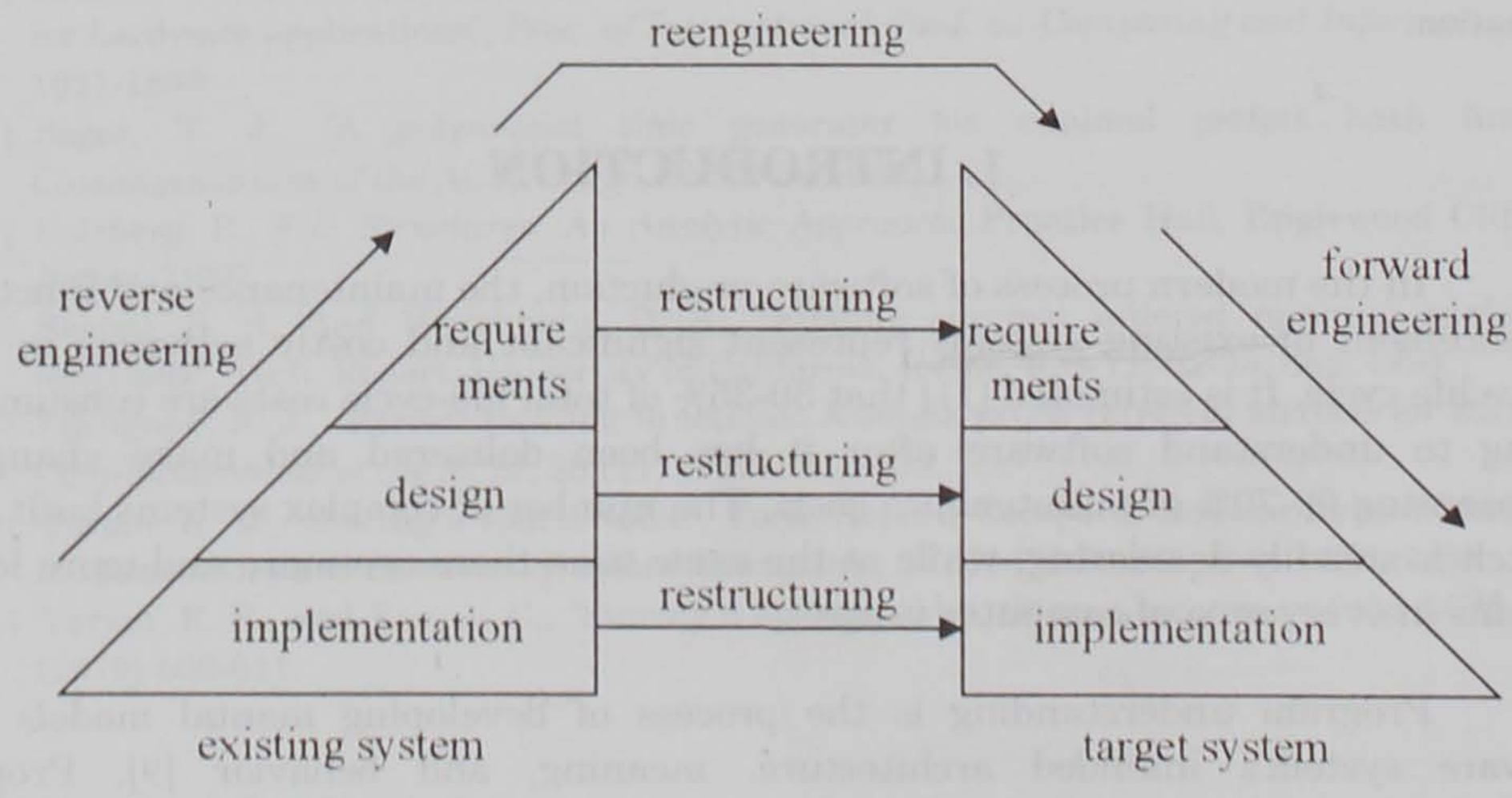

Figure 1: Software reengineering

2.1. Software maintenance

These tools are meant to support the activities of corrective, perfective and adaptive maintenance of large software systems. These tools maintain the central repository of information about a system structure formed in the process of the static analysis of program code. The information may be presented in various forms:

- Cross-reference reports (e.g. the list of all functions that use a certain global variable)
- Various presentations of the system structure (control flow graphs, data flow graphs, call hierarchy graphs, class hierarchy graphs, data dictionaries,...)
- Metric data etc.

Table 1: Selected reverse engineering and reengineering tools

Manufacturer	tool	language	platform	application areas
DBStar, Inc	DBStar	from DB2 into Oracle, Sybase, ...	Sun	database design, data reengineering, business rule extraction
Rational	Rational Rose	C++	Windows, Unix	object-oriented forward and reverse design
Hewlett-Packard	AdaFormat	Ada	HP-UX	reformatting
Reasoning Systems	Software Refinery	Ada, FORTRAN, COBOL, C, ...	HP UX, IBM/AIX, Sun OS	set of tool generators for reverse engineering and reengineering
VERILOG	Logiscope	more that 35	UNIX, VMS, IBM/MVS, ...	metrics, testing - coverage analysis
McCabe	Visual Toolsets	Ada, C, C++, COBOL, Pascal, FORTAN, ...	VAX/VMS, Sun/Sparc, DEC/Ultrix, ...	integrated environment: metrics, restructuring, reusable component extraction,...
Cadre	Ensemble	C	AIX, Sun OS, ULTRIX	Integrated forward and reverse engineering environment
Viasoft, Inc.	Existing Systems Workbench	various COBOL dialects	IBM/MVS, Windows	integrated environment for maintenance, redocumentation and restructuring
General Electric	ENCORE	from FORTRAN into ADA	Sun Sparc	source code translation, restructuring
Xinotech	2001	COBOL	Windows, OS/2, Unix	Analysis and transformations to solve year 2000 problem
Xinotech	Object Abstractor	from FORTRAN COBOL into Ada, SmallTalk, C++	Windows, OS/2, Unix	transformation, objectification, reusable components extraction

Built-in editors typically possess a hypertext-like navigation functionality (e.g. by selecting a function name it is possible to locate the function definition in the source listing). Navigation through graphical presentations of the system structure is also provided.

The functionality of selecting and filtering information is very important because of the tremendous amount of data generated for a large system. Tools typically provide a number of predefined queries, e.g. to display a selected function and its immediate successors in a hierarchy call graph. Some tools (e.g. SMARTSystem) support user defined queries in the information database.

Special attention is paid to maintaining the consistency of the database while changes to the code are being made, by incremental parsing technique and tracking the changes.

Maintenance support tools are frequently integrated with other reengineering tools in maintenance and reengineering environments (Fig. 2 presents an example of such an integration). Interoperability with other development tools (debuggers, emulators, compilers, version control systems, etc.) is usually provided.

2.2. Integrated development and maintenance environments

In recent times the functionality of maintenance tools is being integrated more and more in software development environments as a supplement to the main functionality of compiling, editing and debugging programs. An example of such an integrated environment is Microsoft Visual C++. In these environments, there is relatively modest support for reverse engineering activities - basic structure presentation and navigation functionality.

2.3. Reverse engineering subsystems in design support tools

Tools that support requirement analysis and software design phases achieve various manipulations with specifications (creating, updating, consistency checking and animating) and source code generation based on certain design methodology. More recent tools embody a reverse engineering component that is used to update design specifications after the changes are made to the source code outside the design tool. The scope of this analysis is typically limited to systems with already existing specifications - e.g. the analysis uses the design information built in the code in the form of comments.

2.4. Redocumentation

Redocumentation is the oldest form of reverse engineering. It deals with creating design specifications from the existing source code. The tool output mainly is in the form of a textual document that describes the structural aspects of the system

and is formatted according to some of the standards for software documentation (e.g. MIL_STD 498). Some tools use special comments embedded in code to document the functional aspects of the system. Documentation in electronic form is mainly hypertext-like, and often can be exported in format that is recognized by some design tool.

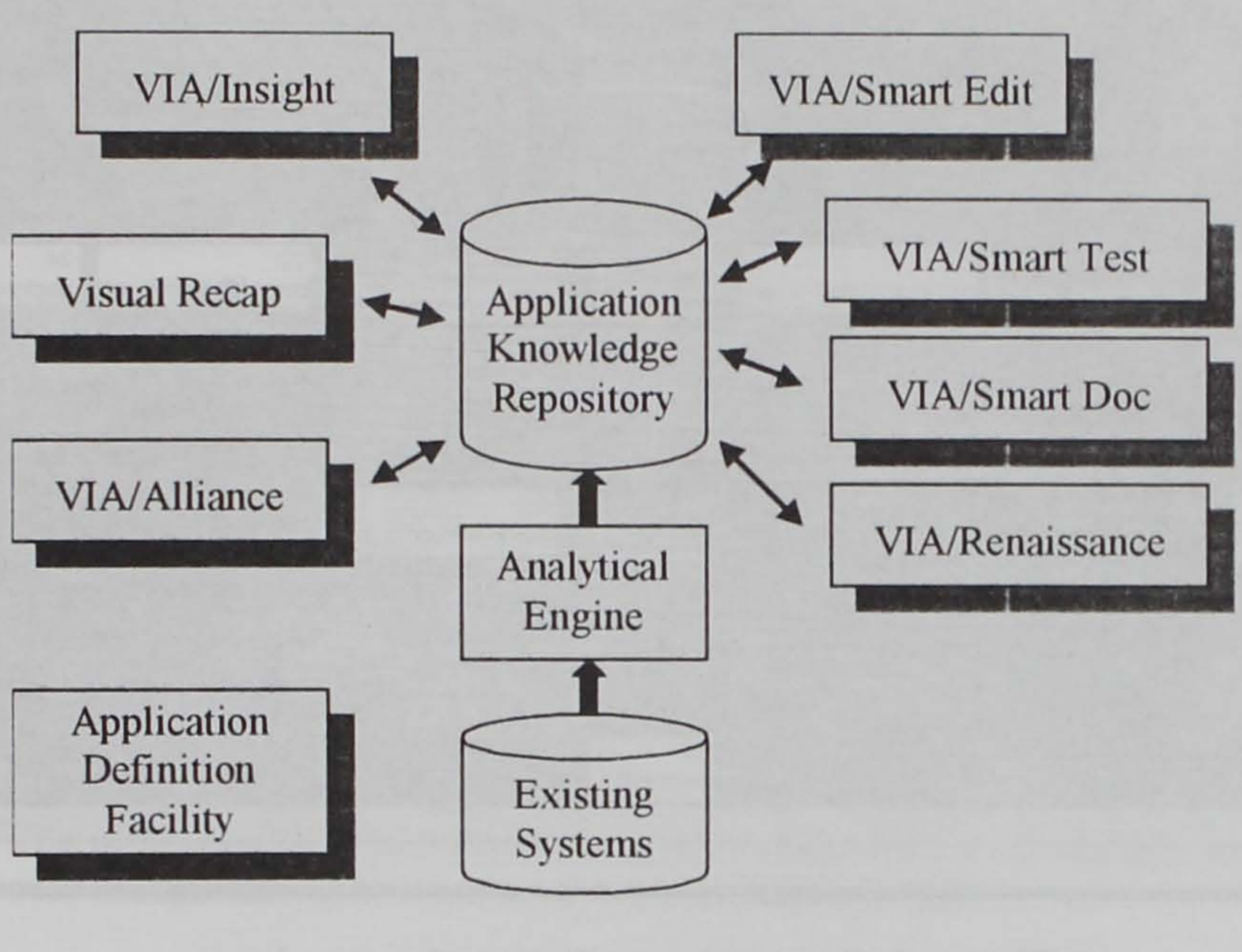

Figure 2: The structure of Existing Systems Workbench integrated environment

2.5. Metric analysis

Numerous metric parameters are defined and used in practice to assess size, complexity, quality, maintability and other code parameters, the impact of change in one part of code to the rest, to identify reusable components [2] etc. For example, McCabe's Visual Quality Toolset calculates 20 different classical and 13 object-oriented metrics. The data is presented in a graphical manner - using a structure graph in which modules are shown in different colors denoting different values for some metric parameters (Fig. 3).

2.6. Restructuring

Restructuring is the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behavior (functionality and semantics).

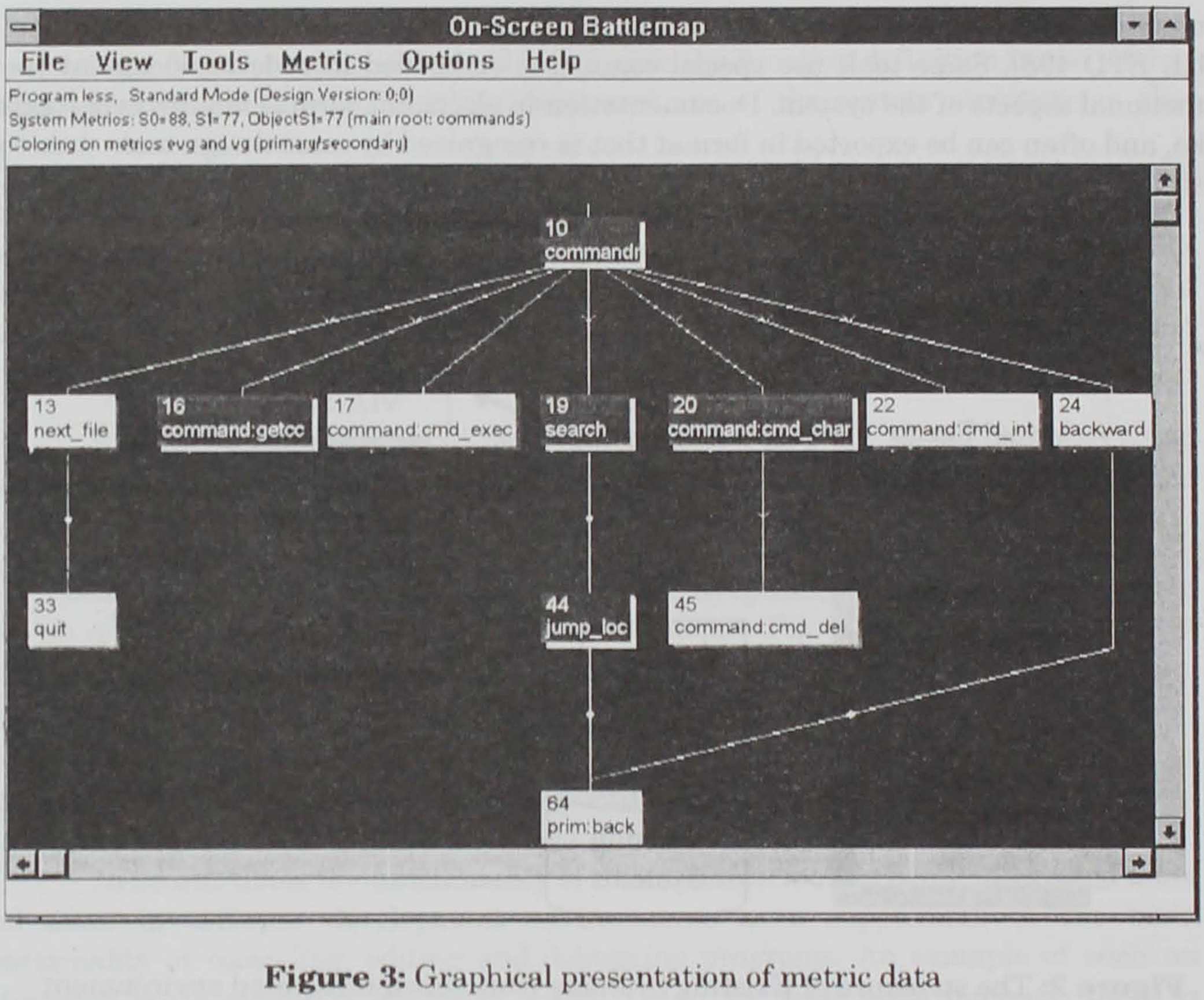

Figure 3: Graphical presentation of metric data

The simplest form of restructuring is code reformatting (pretty-printing) including code indentation, identifier capitalization, keywords marking etc. A significant form of restructuring is source code translation - the transformation from one programming language to another (e.g. form COBOL to ADA) or from one version of some language to another dialect of the same language. Another kind of restructuring is retargeting - migration to another configuration or target platform. Yet another kind of restructuring is name space rationalization - establishing a uniform naming convention for logically same data across various system components.

2.7. Objectification

Program objectification involves transformation of the procedural program in a functionally equivalent program in object-oriented style, into another programming language or into the more recent dialect of the same language. Object-oriented style means that: a) the program structure is defined by the structure of its classes; b) the redundancy of relations between classes is minimal; c) program behavior is determined by methods associated to classes, and d) delayed binding is used to avoid selecting a piece of code statically in cases when there are more variants of implementing some functionality.

For example, Xinotech's Object Abstractor tool supports the process of transforming procedural programs to object-oriented Ada 95 programming language. The process involves restructuring data types and subtype derivation, data grouping, identifying methods and object candidates from program fragments using various strategies and redesigning the resulting packets (Ada's equivalent to classes) to achieve a satisfying level of reusability. The tool embodies a knowledge base in which programming clichés, information abstraction and program transformation rules are stored. The rules are written in XPAL language (Xinotech Plan Abstraction Metalanguage).

2.8. Reusable components extraction

The concept of software reuse concerns the mass production of software components to form a repository from which they can be selected and combined in more complex components or used to develop a new software system [1]. The objective is to achieve a more productive software development process and to improve the quality of software products. The development of a huge component repository from scratch is a great initial investment, and a viable alternative is the extraction of components from existing systems. The concept of reusability applies not only to code, but also to design and architecture solutions.

The process of the extraction of reusable components from existing system is termed reuse reengineering. It consists of the following activities:

- analysis of an existing system to identify candidates for reusable components
- modification of extracted components to decouple them from the rest of the system
- creation of a functional specification for the extracted components

Components are stored in a repository from which they can be selected according to the required functionality. Techniques used to identify components and functionally describe code include structure methods based on metrics, based on programming clichés and formal methods.

2.9. Year 2000 problem (Y2K)

The problem of incorrect manipulation with dates is technically limited, but economically very significant; it is estimated that the worldwide cost of its elimination could approach several hundred billion dollars and the time to solve it is limited. Although the problem mainly affects information systems, it also strikes hardware, operating systems, embedded and communications systems and all other computer systems that operate with dates. The problem involves the following implementation errors:

- Representing the year with a two-digit number.

- Incorrectly calculating leap years.
- Hard-coding the prefix 19_in code, or using the date field to code 'magic numbers' (e.g. 99 means 'never delete this record').
- The overflow in the field for storing a complete date.

There are a number of different techniques to solve these problems depending on whether there is a source code for the system and whether the date format is extendible: extending the date field, coding in the same field but in the binary system, bridging components inside which no change is being made, etc. Modifications are needed not only in data processing modules, but also in interface modules (e.g. date entry fields).

Customized tools to support the elimination of Y2K problems estimate the amount of affected code, locate the program fragments that need change, automatically make changes and verify changed code using standard reverse engineering techniques.

2.10. Meta tools

Meta tools are used to create reverse engineering and reengineering tools. One of the well-known toolsets in this category is Reasoning Systems' Software Refinery. It consists of three tools: DIALECT is used to generate syntax-semantic analyzers for a particular language, REFINE is used to generate a data repository subsystem, and INTERVISTA is used to generate a user interface subsystem (Fig. 4).

3. TOOL SELECTION AND EVALUATION FRAMEWORK

The CASE tools market is in expansion because there is a clearly expressed need in the software industry to use these tools to cut production expenses and improve programmer productivity and software quality. On the other hand, the results from several studies [6] show that there are obstacles in adopting this technology in practice. A survey of several hundred software companies has shown that less than 25% of personnel uses any tool. A year after introduction, 70% of the tools are not used at all, 25% of the tools are used by a small percent of the staff, and only 5% of the tools are widely used, but not in their full capacity. This is mainly caused by mistakes made when selecting appropriate tools and by underestimating the effect of introducing new technology. The learning curve shows that a period of 6 to 12 months is needed for productivity to gain the same level as before the introduction of a new technology, and productivity improves only after that period.

Far too often, tool assessments are completed by the product's vendor or by someone who:

- just scans the brochures and user manual,
- is unfamiliar with the tool's methods,

- lacks an understanding of the project or user requirements, or
- uses the tool for a 30-day trial on useless examples that fall short of testing the tool's functionality.

To eliminate these problems, the Software Engineering Institute of Carnegie Mellon University and the Westinghouse Software Tools Evaluation Committee have established a general framework for CASE tool evaluation and adoption processes [3], [8]. In this section, the specific criteria for the evaluation of reverse engineering tools are discussed, in accordance to the general framework.

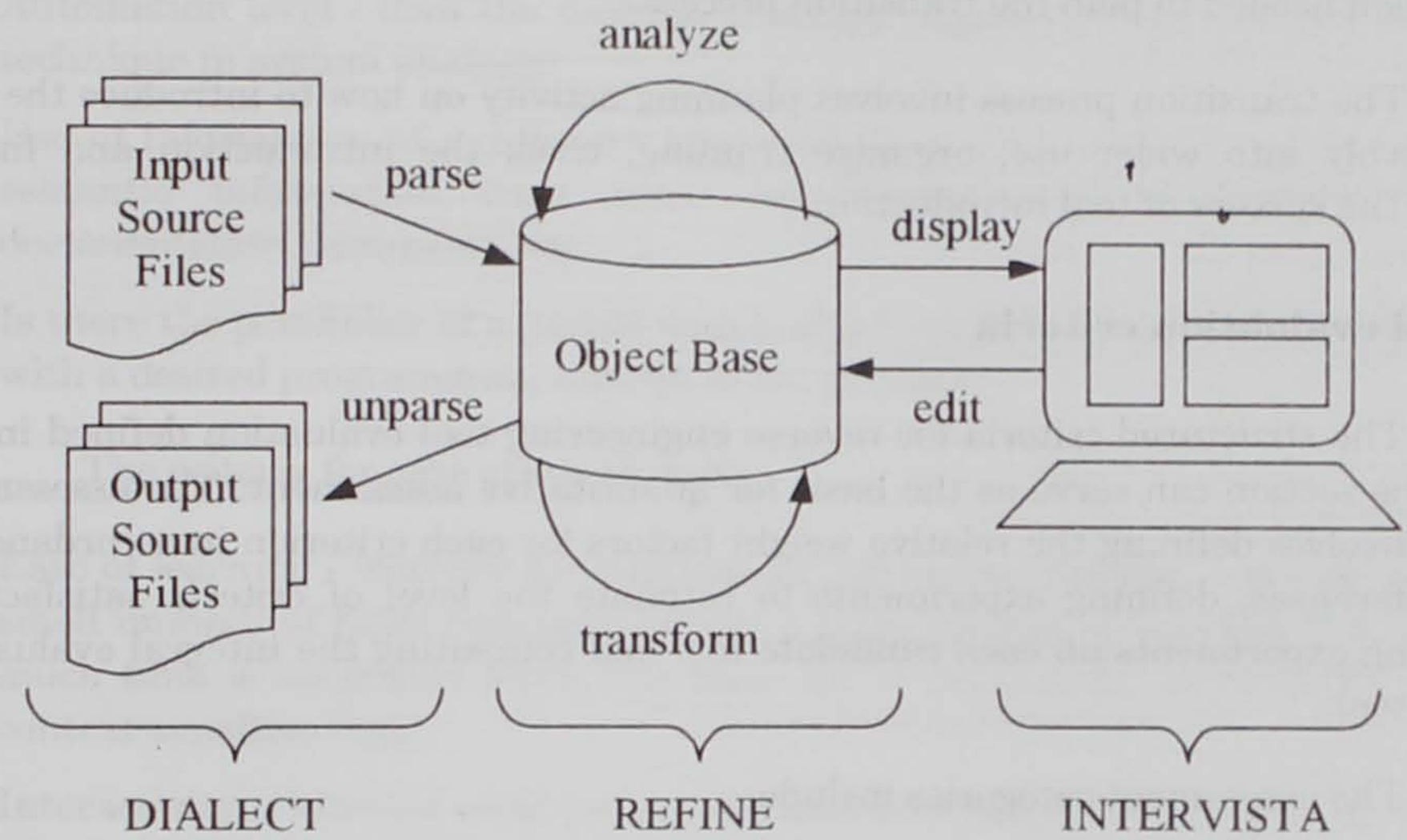

Figure 4: Software Refinery toolset

The process of introducing a new tool into regular use is composed of the following activities:

- preparation
- evaluation and selection
- pilot project
- transition

During the preparation activity, the general objectives and organizational aspects of the whole process are established: resources and process dynamics. The evaluation and selection activity is performed in the following phases:

- Needs analysis, to establish the purpose for which the tool will be used.
- Analysis of the existing environment to establish various technical, economical and other limitations in tool selection.

- Establishing a list of candidates and their categorization, that is, determining their functionality.
- Evaluation of candidates; the assessment criteria and methods for their estimation are defined.
- Tool selection on the basis of a integral criterion.

The purpose of the pilot project is to provide a realistic trial of the selected tool in the target environment. That should be a project of limited value and duration, with typical requirements. It should confirm the usability of the tool and generate information needed to plan the transition process.

The transition process involves planning activity on how to introduce the tool progressively into wider use, organize training, track the introduction and finally estimate the success of tool introduction.

3.1. Tool evaluation criteria

The structured criteria for reverse engineering tool evaluation defined in the rest of the section can serve as the basis for quantitative assessment. The assessment process involves defining the relative weight factors for each criterion in accordance to user preferences, defining experiments to estimate the level of criteria satisfaction, conducting experiments on each candidate tool and computing the integral evaluation for each tool.

The assessment categories include:

- functionality
- ease of use
- extensibility
- robustness
- environment fitting
- level of support

The criteria of functionality include:

- Types of activities that are supported by the tool (design recovery, redocumentation, reengineering, ...)
- The level of abstraction: does the tool support a system modeled on a structural, design or functional level?
- Support for multiple knowledge domains: does the tool support multiple programming languages and application domains (e.g. databases and embedded systems). Based on this aspect, there are: a) domain specific solutions (suitable for one particular domain, not applicable to others); b) flexible or retargetable

solutions (that can be adapted by the user to another domain, e.g. by adding a new language parser), and c) general solutions that can be used for multiple domains without the need for modifications.

- Assessment of the underlying methodology

1. The number of different analysis techniques used by the tool.
2. Does the tool analyze static and/or dynamic aspects of the system?
3. Scalability - the ability to analyze both small and big systems equally efficiently. What is the biggest system that can be analyzed with acceptable tool performance?
4. Automation level - does the tool use a manual, semiautomatic or fully automatic technique in system analysis?
5. Use of information of a different kind - whether the tool uses, besides syntactic-semantic information from code, other sources of information such as documentation, comments, etc.
6. Is there the possibility of a partial match of information in cases when a full match with a desired programming concept is not possible?

The criteria for ease of use include:

- Ease of learning - whether the tool has an interactive learning support; whether a small number of basic commands cover a large percent of tool functionality; how much time is needed to learn this basic set of commands; whether the tool has context-sensitive help.
- Interactivity - whether easy navigation and selection of information is provided; whether the user interface includes graphics, sound, icons.
- Customizability - whether the main elements of the user interface are customizable.
- Active support - whether the tool caches most frequent user operations; is it passive, so that actions are initiated by the user only, or active, so that some actions are suggested by the tool.
- Workgroup support in a centralized and distributed environment - are there mechanisms for the transparent use of distributed information? Is there configuration control and version control support?
- Polymorphism of commands - whether the tool has a large number of commands with specific labels or a limited number with clear semantics that can operate on various kinds of objects and collections of diverse objects. For example, whether there are separate commands for copying a graph and copying a text, or the same copy command can be used for a selection that contains both graph and text).
- Predictability and error correction - whether there is a warning preceding the execution of all potentially dangerous commands; whether the undo operation is supplied.

The criteria for extensibility include:

- The possibility of supporting conceptual domains which are not initially built-in.
- The possibility of supporting analysis methods which are not initially built-in (e.g. adding a new language parser).
- The possibility of supporting a new kind of information presentation.
- The possibility of adding new functionality (open software architecture).

The criteria for robustness include:

- Tolerance to errors in the input data and detection of inconsistency in internal data caused by some external action (e.g. editing a source file by some external editor).
- Analysis exactness and compatibility with the standards of the methodology used (e.g. whether the tool correctly calculates pointer aliasing information)
- Low level of failures and self-instrumentation - built-in mechanisms of self-testing and logging of the failure situations.
- Vertical compatibility between different versions of the tool; whether a tool can use old version data; whether a new and old version can coexist at the same time in the system.

The criteria for environment fitting include:

- The use of methodology, presentation and vocabulary already known to the user.
- The command set of the tool should not be in conflict with command sets of other tools used by the user (e.g. commands with the same name and a different behavior).
- Tool availability on the user's hardware/software platform; operation with the satisfactory performance level; easy installation.
- Interoperability, that is, the tool's ability to exchange data and cooperate in other ways with other tools. Generally, there are strongly coupled environments where tool interaction is programmed into each tool, and loosely coupled environments where standard data formats and communication mechanisms are used by each tool.

The criteria for the level of support include:

- The history of the tool and the producer's reputation - whether the tool is known and mature and there are known uses in areas close to user needs; whether the future of the producer and tool support is ensured.
- Is there a possibility of obtaining the source code? Are there any possible limitations on using products created by the tool?
- Is there enough support for installing, training, on-line troubleshooting, and maintenance? At what rate are new versions released?

3.2. Quantitative evaluation

We shall now highlight the evaluation process. A brief evaluation is made when time constraints preclude a detailed evaluation or when a credible, expert user can effectively summarize the tool's usefulness or uselessness. It answers the question *how well* the tool performs. The brief report has to include the tool's good and bad points and comments about the nature of the evaluation and any pertinent, crucial points about the product or vendor.

Quantitative assessment is performed when we need a detailed or comparative analysis of one or more tools. To make sure the assessments are fair, several assessment practitioners should be assigned to the same tool. To establish credibility, the tools should be applied on scaled down versions of real projects, thus obtaining a more realistic assessment of what the tool can actually do.

The assessment instrument A is the set of questions q_j^{Ci}, based on criteria from the previous section, categorized in six categories C_i plus one additional category for other specific criteria not covered by the existing six categories:

$$A = \bigcup_{1 \le i \le 7} C_i \ , \ C_i = \{q_1^{Ci}, q_2^{Ci}, q_{ni}^{Ci}\}$$

where n_i is the number of questions in category C_i. Each assessment practitioner should be trained on how to use the weighted assessment instrument and is assigned a set of tools to classify and evaluate. Each category (not a particular question) is assigned a relative weight W_{C_i} in percents, based on the user's requirements, for example, to emphasize functionality by weighting its questions 30% over robustness (5%).

The practitioner then evaluates each tool, giving one of three possible scores $S(q_j^{C_i}) \in \{0, 5, 10\}$ for each question $q_j^{C_i}$ to limit the amount of subjectivity. The weighted score $S(C_i)$ in each category C_i, and the overall score $S(A)$ are obtained as follows:

$$S(C_i) = W_{C_i} \sum_{1 \le j \le ni} S(q_j^{C_i}) \ , \ S(A) = \sum_{1 \le i \le 7} S(C_i)$$

The maximum score $S_{\max}(C_i)$ in each category C_i is the score a tool can achieve if it receives the maximum points allowed on each question in C_i. The median score $S_{\text{med}}(C_i)$ in each category C_i is the arithmetic average of scores $S(C_i)$ in category C_i of all tools that are currently being evaluated.

Category scores are plotted as a line graph, like the one in Fig. 5, to provide a pictorial representation of the result. Figure 5 shows that Tool X scored a little above the median in ease of use, robustness, ease of insertion, and "other." In this case, the

The criteria for extensibility include:

- The possibility of supporting conceptual domains which are not initially built-in.
- The possibility of supporting analysis methods which are not initially built-in (e.g. adding a new language parser).
- The possibility of supporting a new kind of information presentation.
- The possibility of adding new functionality (open software architecture).

The criteria for robustness include:

- Tolerance to errors in the input data and detection of inconsistency in internal data caused by some external action (e.g. editing a source file by some external editor).
- Analysis exactness and compatibility with the standards of the methodology used (e.g. whether the tool correctly calculates pointer aliasing information)
- Low level of failures and self-instrumentation - built-in mechanisms of self-testing and logging of the failure situations.
- Vertical compatibility between different versions of the tool; whether a tool can use old version data; whether a new and old version can coexist at the same time in the system.

The criteria for environment fitting include:

- The use of methodology, presentation and vocabulary already known to the user.
- The command set of the tool should not be in conflict with command sets of other tools used by the user (e.g. commands with the same name and a different behavior).
- Tool availability on the user's hardware/software platform; operation with the satisfactory performance level; easy installation.
- Interoperability, that is, the tool's ability to exchange data and cooperate in other ways with other tools. Generally, there are strongly coupled environments where tool interaction is programmed into each tool, and loosely coupled environments where standard data formats and communication mechanisms are used by each tool.

The criteria for the level of support include:

- The history of the tool and the producer's reputation - whether the tool is known and mature and there are known uses in areas close to user needs; whether the future of the producer and tool support is ensured.
- Is there a possibility of obtaining the source code? Are there any possible limitations on using products created by the tool?
- Is there enough support for installing, training, on-line troubleshooting, and maintenance? At what rate are new versions released?

The criteria for extensibility include:

- The possibility of supporting conceptual domains which are not initially built-in.
- The possibility of supporting analysis methods which are not initially built-in (e.g. adding a new language parser).
- The possibility of supporting a new kind of information presentation.
- The possibility of adding new functionality (open software architecture).

The criteria for robustness include:

- Tolerance to errors in the input data and detection of inconsistency in internal data caused by some external action (e.g. editing a source file by some external editor).
- Analysis exactness and compatibility with the standards of the methodology used (e.g. whether the tool correctly calculates pointer aliasing information)
- Low level of failures and self-instrumentation - built-in mechanisms of self-testing and logging of the failure situations.
- Vertical compatibility between different versions of the tool; whether a tool can use old version data; whether a new and old version can coexist at the same time in the system.

The criteria for environment fitting include:

- The use of methodology, presentation and vocabulary already known to the user.
- The command set of the tool should not be in conflict with command sets of other tools used by the user (e.g. commands with the same name and a different behavior).
- Tool availability on the user's hardware/software platform; operation with the satisfactory performance level; easy installation.
- Interoperability, that is, the tool's ability to exchange data and cooperate in other ways with other tools. Generally, there are strongly coupled environments where tool interaction is programmed into each tool, and loosely coupled environments where standard data formats and communication mechanisms are used by each tool.

The criteria for the level of support include:

- The history of the tool and the producer's reputation - whether the tool is known and mature and there are known uses in areas close to user needs; whether the future of the producer and tool support is ensured.
- Is there a possibility of obtaining the source code? Are there any possible limitations on using products created by the tool?
- Is there enough support for installing, training, on-line troubleshooting, and maintenance? At what rate are new versions released?

3.2. Quantitative evaluation

We shall now highlight the evaluation process. A brief evaluation is made when time constraints preclude a detailed evaluation or when a credible, expert user can effectively summarize the tool's usefulness or uselessness. It answers the question *how well* the tool performs. The brief report has to include the tool's good and bad points and comments about the nature of the evaluation and any pertinent, crucial points about the product or vendor.

Quantitative assessment is performed when we need a detailed or comparative analysis of one or more tools. To make sure the assessments are fair, several assessment practitioners should be assigned to the same tool. To establish credibility, the tools should be applied on scaled down versions of real projects, thus obtaining a more realistic assessment of what the tool can actually do.

The assessment instrument A is the set of questions q_j^{Ci}, based on criteria from the previous section, categorized in six categories C_i plus one additional category for other specific criteria not covered by the existing six categories:

$$A = \bigcup_{1 \le i \le 7} C_i \,,\; C_i = \{q_1^{Ci}, q_2^{Ci}, q_{ni}^{Ci}\}$$

where n_i is the number of questions in category C_i. Each assessment practitioner should be trained on how to use the weighted assessment instrument and is assigned a set of tools to classify and evaluate. Each category (not a particular question) is assigned a relative weight W_{C_i} in percents, based on the user's requirements, for example, to emphasize functionality by weighting its questions 30% over robustness (5%).

The practitioner then evaluates each tool, giving one of three possible scores $S(q_j^{C_i}) \in \{0, 5, 10\}$ for each question $q_j^{C_i}$ to limit the amount of subjectivity. The weighted score $S(C_i)$ in each category C_i, and the overall score $S(A)$ are obtained as follows:

$$S(C_i) = W_{C_i} \sum_{1 \le j \le ni} S(q_j^{C_i}) \,,\; S(A) = \sum_{1 \le i \le 7} S(C_i)$$

The maximum score $S_{\max}(C_i)$ in each category C_i is the score a tool can achieve if it receives the maximum points allowed on each question in C_i. The median score $S_{\text{med}}(C_i)$ in each category C_i is the arithmetic average of scores $S(C_i)$ in category C_i of all tools that are currently being evaluated.

Category scores are plotted as a line graph, like the one in Fig. 5, to provide a pictorial representation of the result. Figure 5 shows that Tool X scored a little above the median in ease of use, robustness, ease of insertion, and "other." In this case, the

assigned weights clearly show that functionality is the top priority, followed by ease of use.

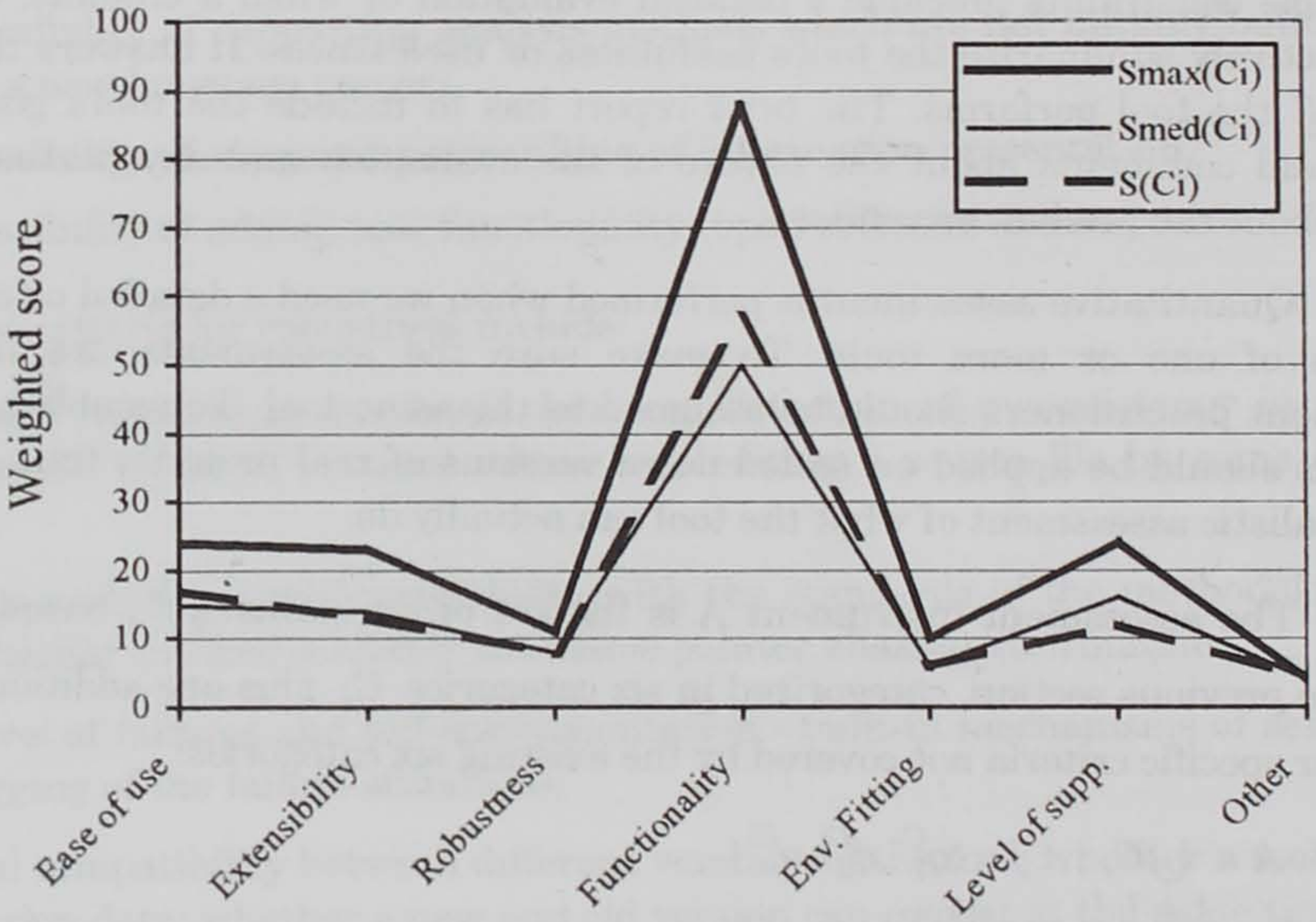

Figure 5: Sample tool evaluation graph

In the final step, the evaluator takes the results of the assessment, extracts the critical and essential characteristics according to the user's requirements, and completes a tailored summary of what the score really represents. Table 2 shows the tailored summary of Tool X in an abbreviated format.

Table 2: Tailored summary of tool X

Category	$S_{max}(C_i)$	$S_{med}(C_i)$	$S(C_i)$	Comments
Ease of use	24	15	17	Has good user interface, but keys can t be tailored...
Extensibility	23	14	13	Could be upgraded to network version...
Robustness	11	8	8	There is vertical compatibility between versions...
Functionality	88	50	57	Supports the methodology well, but does not do...
Environment Fitting	10	6	6	Has good installation procedures, but is available only for...
Level of support	24	16	12	There is maintenance, but no hotline...
Other	4	4	4	High cost...
Overall score	$S_{max}(A) = 184$	$S_{med}(A) = 113$	$S(A) = 117$	

4. CONCLUSIONS

While there is a great need for automating arduous and costly maintenance and reengineering activities, there are also obstacles to adopting new tools and techniques. Frequently, potential users are not aware that there exists automated support for the kinds of tasks they perform. Therefore, we tried to increase this awareness by enumerating the application areas for reverse engineering tools in the first part of this paper. Other common factors that impede tool adoption include unsystematic and superficial tool selection and evaluation, which results in buying an inappropriate tool. Hence, we are presenting a framework for quantitative tool evaluation that for the most part eliminates subjectivity from the evaluation process, thus increasing the probability of selecting the appropriate tool that will actually fit the needs of the user.

Tool evaluation can continue after the tool has been introduced into regular use by collecting and comparing the productivity and quality statistics of the reengineering process, before and after the tool's introduction, to estimate whether it lives up to expectations.

REFERENCES

[1] Ahrens, J. D., and Prywes, N. S., "Transition to a legacy- and reuse-based software life cycle", *IEEE Computer*, October 1995, 27-36.

[2] Caldiera, G., and Basili, V. R., "Identifying and qualifying reusable software components", *IEEE Computer*, February 1991, 61-69.

[3] Firth, R. et al, "A guide to the classification and assessment of software engineering tools", Technical Report CMU/SEI-87-TR-10, Software Engineering Institute, Carnegie Mellon University, 1987.

[4] Fuggetta, A, "A classification of CASE technology", *IEEE Computer*, December 1993.

[5] Information Technology - "Guide for ISO/IEC 12207 (Software life cycle processes)", Draft Tech. Rpt. PDTR 15271, ISO/IEC JTC1/SC7/WG7 N94, International Organization for Standardization, 1996.

[6] Kemerer, C. F., "How the learning curve affects CASE tool adoption", *IEEE Software*, May 1992, 23-29.

[7] Lejter, M., Meyers, S., and Reiss, S. P., "Support for maintaining object-oriented programs", *IEEE Transactions on Software Engineering*, 18 (12) (1992) 1045-1052.

[8] Mosley, V., "How to assess tools efficiently and quantitatively", *IEEE Software*, May 1992, 29-32.

[9] Müller, H. A., "Understanding software systems using reverse engineering technologies research and practice", A tutorial presented at ICSE-18 18th International Conference on Software Engineering, Berlin, Germany, March 25-29, 1996.

[10] Olsem, M.R., and Sittenauer, C., *Reengineering Technology Report*, Software Technology Support Center, OO-ALC/TISEC, 7278 Fourth Street, Hill AFB, UT 84056-5205, 1995.

[11] Tilley, S. R., and Smith, D. B., *Perspectives on Legacy System Reengineering*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213-3890, 1995.