MITSUBISHI ELECTRIC RESEARCH LABORATORIES
http://www.merl.com

Knowledge-Intensive Software Engineering Tools

Charles Rich, Richard C. Waters

TR91-03 September 1991

Abstract

Essentially all current software engineering tools share a common technological approach: They use a shallow representation of software objects and manipulate this representation using procedural methods. This approach has the benefit that it allows one to get off to a fast start and quickly provide a tool that delivers benefits. In addition, software engineering tools can undoubtedly be extended to a considerable extent within this approach. However, the approach will eventually reach a point of diminishing returns where more knowledge-intensive approaches will be needed to achieve significantly higher levels of capability. We believe that the software engineering tools of the future will have to rely on deep representation, inspection methods, and intelligent assistance. Deep representation will be necessary to capture a sufficiently large part of knowledge about programming in general and particular programs. Inspection methods (recognizing standard solutions rather than reinventing them) will be necessary to deal with complexity. Intelligent assistance will be necessary, because complete automation is not a realistic possibility in the foreseeable future, rather only parts of the programming process can be automated.

Mitsubishi Electric Research Laboratories

Technical Report 91-03 September 30, 1991

Knowledge-Intensive Software Engineering Tools

by

Charles Rich
Richard C. Waters

Abstract

Essentially all current software engineering tools share a common technological approach: They use a shallow representation of software objects and manipulate this representation using procedural methods. This approach has the benefit that it allows one to get off to a fast start and quickly provide a tool that delivers benefits. In addition, software engineering tools can undoubtedly be extended to a considerable extent within this approach. However, the approach will eventually reach a point of diminishing returns where more knowledge-intensive approaches will be needed to achieve significantly higher levels of capability.

We believe that the software engineering tools of the future will have to rely on deep representation, inspection methods, and intelligent assistance. Deep representation will be necessary to capture a sufficiently large part of knowledge about programming in general and particular programs. Inspection methods (recognizing standard solutions rather than reinventing them) will be necessary to deal with complexity. Intelligent assistance will be necessary, because complete automation is not a realistic possibility in the foreseeable future, rather only parts of the programming process can be automated.

Submitted to *IEEE Transactions on Knowledge and Data Engineering*, September 1991.

201 Broadway
Cambridge Massachusetts 02139

Publication History:-

1. First printing, TR 91-03, September 1991

The idea of computer-aided software engineering—applying the power of computers to the software process itself—has been around since the first programmers realized that programming is hard. Rapid early progress came from the introduction of assemblers and then high-level language compilers in the 1950s and early 60s. Each move to a higher level language resulted in a dramatic decrease in the program size needed for a given problem. This in turn yielded dramatic improvements in productivity by simplifying every aspect of the software process.

Stepping up again to a very high-level language has been a goal of computer science since the late 1960s. Unfortunately, there has been little success thus far in automatically compiling such languages into efficient machine code. Overall, although there have been a number of important advances in general-purpose programming languages over the past 20 years, none has had as dramatic an effect as the initial introduction of high-level languages.

With diminishing returns from work on general-purpose programming languages, two less general approaches to automation came to the fore in the 1970s and early 80s: domain-specific and software-task-specific support. Given a sufficiently narrow application domain, such as report generation or employee payroll, it has proven quite feasible to develop a specialized very high-level language (often called a fourth-generation language) and a program generator that compiles it into efficient machine code. It has also proven feasible to develop software engineering tools that (partially) automate specific software tasks, such as program testing or constructing a consistent version of a system.

More recently, a new movement has developed under the rubric of Computer-Aided Software Engineering (CASE). In addition to increasing the power of individual software engineering tools, this approach emphasizes the enterprise-wide integration of software support based on a central on-line repository for software objects, including requirements, designs, and source code. A general-purpose and comprehensive software engineering environment supporting the integrated evolution of all software objects would be an advance as dramatic and pervasive as the move to higher level languages.

Our view of the current status of software engineering tools is indicated by the lower curve in Figure 1. Rapid progress is being made. However, the day is approaching when the power of software tools will be limited by the technology now employed. Some tools, such as program editors, are close to these limits already. Other tools, such as repositories, still have room for large improvement using current technology. (For a sampling of current developments in software tools, see [2, 3].)

After a brief review of the technology underlying current software engineering tools, we discuss the technologies that we feel are essential to support knowledge-intensive tools that are fundamentally more power than today's shallow tools (see the upper curve in Figure 1). The main body of the paper is devoted to a description of an experimental knowledge-intensive tool based on these technologies.

Limits of Current Technology

Essentially all current software engineering tools share a common technological approach: They use a *shallow representation* of software objects and manipulate this representation using *procedural methods*.

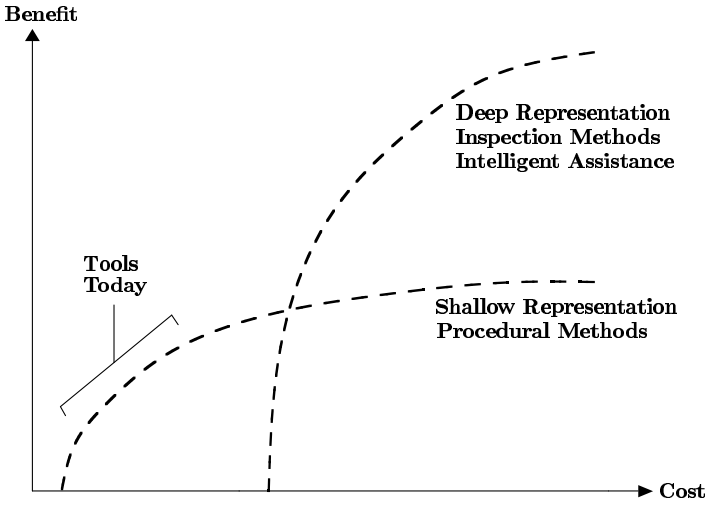

Figure 1: A comparison of current software engineering tool technology (lower curve) with the technology required for knowledge-intensive tools (upper curve).

Shallow representation. For a software engineering tool to operate on a given software object, it must represent the object on line in some fashion. The choice of representation involves a trade-off between the power of the representation and the cost of implementing it. Generally speaking, the shallower a representation, i.e., the less information it represents explicitly, the easier it is to implement. However, the less explicitly information is represented, the harder it is for a tool to manipulate it. For instance, it is next to impossible for a tool that represents requirements merely as textual documents to detect logical inconsistencies in them.

Many current software tools operate at the level of text, parse trees, or diagrams annotated with text. These relatively shallow representations leave much information either implicit or totally absent. For instance, representing a program as a parse tree makes its syntactic structure explicit, but leaves implicit the data- and control-flow structure of the underlying algorithm. As a result, the typical syntax-directed program editor provides good support for syntactic changes. Single commands suffice to add or delete syntactic units and the syntactic correctness of the result can be checked. However, even a conceptually simple algorithmic change typically requires a sequence of disconnected syntactic edits. Also, the editor has no basis for checking the algorithmic correctness of the result.

Some tools for systems analysis use a deeper level of representation. Rather than merely manipulating diagrams, they represent the underlying information using data-dictionary and entity-relationship technology. This allows more powerful editors to be developed and a deeper level of consistency checking.

In general, it makes sense to represent only as much information as a given tool can take advantage of. However, current tools will eventually reach the limits of what they can do with the representations they use.

Procedural methods. For a software engineering tool to support a given task, it

must have knowledge about the task. In current tools, this knowledge is almost exclusively procedural. Without making use of any explicit representation of task knowledge, the tool is simply written so that it performs the task.

Procedural methods have the advantage of enabling you to get off to a quick start. It is typically easy to support the first round of capabilities. However, as more and more features are added, procedural systems become progressively more difficult to modify. For example, it is relatively easy to write a program generator for a simple fourth-generation language. However, as the language is expanded, the program generator rapidly balloons into an unmaintainable monster.

Supporting Knowledge-Intensive Tools

If software engineering tools are to become dramatically more powerful than the tools available today, they must become *knowledge-intensive*. In contrast to current tools, which attempt to leverage off of small amounts of knowledge about software artifacts and software tasks, they must contain large amounts of knowledge about software artifacts and software tasks. In our opinion, this requires the introduction of three key concepts: *deep representation*, *inspection methods*, and *intelligent assistance*.

Deep representation. A software tool cannot reason about information that it does not represent. If a tool is to provide comprehensive support for a complex software task that requires deep understanding of a software artifact, then it must represent the artifact in such a way that all of the relevant information is available to it.

A tool that wishes to support complex algorithmic changes in a program has to have detailed algorithmic information available to it. For instance, in our work we use a representation called the *Plan Calculus* [6] to represent programs and their designs. As compared to program text or parse trees, the Plan Calculus makes data flow, control flow, and the design of a program more explicit. This facilitates the direct manipulation of these features.

To represent nonalgorithmic aspects of software, such as performance requirements, one can use standard artificial-intelligence knowledge-representation and reasoning techniques. For example, the Requirements Apprentice, described in the next section, uses frames and constraints to detect inconsistency and incompleteness in informal requirements descriptions.

An important benefit of deeper representations is that they allow a deeper level of integration between tools. A shared semantic framework enables the incremental exchange of information between tasks necessary to support a more evolutionary software process.

Inspection methods. Software engineers, like engineers in other disciplines, seldom reason from first principles. Rather, they rely whenever possible on their experience with standard building blocks (or *clichés*). Given knowledge of the clichés in a particular application area, it is possible to perform many software engineering tasks *by inspection*. For example, in analysis by inspection, properties of a program are deduced by recognizing occurrences of clichés and referring to their known properties. Similarly, in design by inspection, recognition of key properties of the specification leads to the construction of a design by combining standard design components.

Automated tools are undoubtedly better suited to the tedious task of reasoning from first principles than human software engineers. However, as the size of a problem rises, the complexity of reasoning about it from first principles explodes exponentially. Therefore, if software tools are to automate significant parts of complex software tasks, they too will have to rely on inspection methods.

An important collateral advantage of inspection methods is that clichés lend themselves to declarative (nonprocedural) representation. This has two effects. First, the same library of clichés can be used in support of more than one task, e.g., design and analysis. Second, declarative knowledge is easier to extend. Formalizing suites of clichés and integrating them into an existing library is not trivial. However, it is much easier than modifying a collection of complex interacting procedures.

Intelligent assistance. Even with deep representation and inspection methods, it will not be possible (at least in the foreseeable future) to totally replace software engineers. However, the next generation of tools can aim for intelligent assistance.

Rather than simply accepting and executing commands, an intelligent assistant can check the reasonableness of decisions, fill in missing details, and request advice about how to carry out complex operations. These abilities can contribute to both an engineer's productivity and the reliability of the final software product.

Another hallmark of an intelligent assistant is the ability to explain its actions and decisions in terms that an engineer can understand. This allows engineers to check what the tool has done. It also allows the tool to describe the problems it has encountered when it asks for advice.

Using inspection methods makes it easier for the engineer to understand what the tool is doing and for the tool to understand advice given by the engineer. The names of clichés provide the essential vocabulary for communication between the engineer and the assistant, just as they form the essential vocabulary for communication between human engineers.

The Requirements Apprentice

A central goal of our work has been to develop technologies for knowledge-intensive computerized support of software engineering. Our long term aim has been to create a *Programmer's Apprentice* that can provide powerful support for all aspects of the software process [6]. As steps toward the full apprentice, we have created a series of demonstration systems that support particular software tasks. The latest of these demonstration systems (the Requirements Apprentice) is a particularly good example of what a knowledge-intensive software tool based on deep representation, inspection methods, and intelligent assistance can do.

The focus of the Requirements Apprentice (RA) is on the *formalization* phase that bridges the gap between an informal and formal specification. This is a crucial area of weakness in the current state of the art.

Figure 2 shows the role of the RA in relation to other agents involved in the software process. Note that the RA does not interact directly with an end-user, but rather is an intelligent assistant to a requirements analyst.

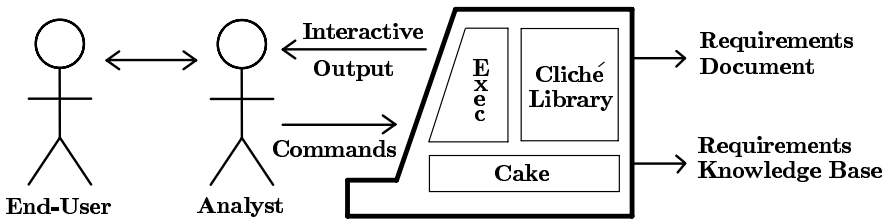

Figure 2: The Requirements Apprentice.

The RA produces three kinds of output. Interactive output notifies the analyst of conclusions drawn and inconsistencies detected while requirements information is being entered. A machine-manipulable *Requirements Knowledge-Base* (RKB) is a representation of everything the RA knows about an evolving requirement. (In the long term, it is intended that the RKB be accessed directly by other tools.) Finally, the RA can create a more or less traditional requirements document summarizing the RKB.

Figure 2 shows that the RA is composed of three modules. *Cake* [5, 6] is a knowledge-representation and reasoning system, which supports the reasoning abilities of the RA (and the rest of the Programmer's Apprentice). Cake provides basic facilities for propositional deduction (including the detection of contradictions), reasoning about equalities, maintenance of dependencies between deduced facts, and incremental retraction of previously asserted facts. It also provides the basis for a deep representation of requirements knowledge.

The *executive* handles interaction with the analyst and provides high-level control of the reasoning performed by Cake. The analyst communicates with the executive by issuing commands. Each command provides fragmentary information about an aspect of the requirement being specified. The immediate implications of a command are processed by Cake, added to the RKB, and checked for consistency. If the analyst makes a change in the description (to correct an inconsistency or simply due to a change of mind) the executive incrementally incorporates this modification into the RKB, retracting invalidated deductions and replacing them with new deductions.

The *cliché library* is a declarative repository of information relevant to requirements in general and to domains of particular interest. It forms the basis for applying inspection methods to requirements. When creating a requirement, the information unique to the particular problem comes from the analyst. However, the bulk of general information about the domain comes from the cliché library. The structure of the library is illustrated in Figure 3.

Much of our experimentation with the RA has revolved around the hypothetical library-management-system requirement that has been used as an example in the International Workshops on Software Specification and Design [10]. Three examples of clichés in this area are *repository*, *information system*, and *tracking system*.

A repository is an entity in the physical world. The basic function of a repository is to ensure that items entering the repository will be available for later removal. There are several kinds of repositories depending on whether or not the items are intended to be returned and whether or not the items are grouped into classes. Example repositories

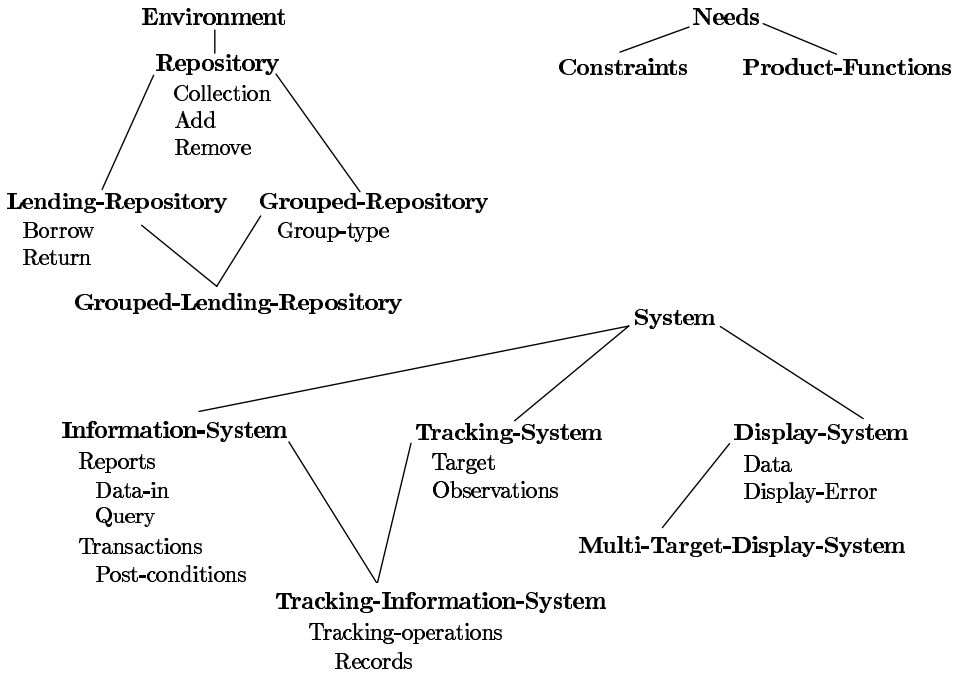

Figure 3: The structure of a fragment of the requirements cliché library.

include a storage warehouse (simple repository for unrelated items), a grocery store (simple repository for items grouped in classes), and a rental car agency (lending repository for items grouped in classes).

In contrast to the repository cliché, the *information system* cliché describes a class of programs rather than a class of physical objects. The intent of the information system cliché is to capture the commonality between programs such as personnel systems, bibliographic databases, and inventory control systems. The central roles of an information system are a set of *reports* that display parts of the data being stored and a set of updating *transactions* that create/modify/delete the data.

A *tracking system* keeps track of the state of some *target* object. It does this by making *observations*. The observations can either sense the state of the object directly (e.g., the way a radar keeps track of the position of airplanes) or observe the operations that change the state (e.g., the way a set of turnstiles keeps track of the number of people in a building). The *tracking information system* cliché combines the features of an information system and a tracking system.

The Requirements Apprentice in action. The transcript in Figure 4 shows the RA being used to construct a requirement for the library management system example. It is part of a larger transcript presented in [4].

The first three commands in Figure 4 begin the process of building a requirement by introducing the key high level terms. The quotes around the word library in the RA's response to the third command indicate that the RA does not know what a library is.

To operate in the intended manner, the RA must have a considerable amount of background knowledge relevant to the requirement at hand. However, it is unrealistic

```
1> (Find-Requirement Library-System)
Beginning-A-New-Requirement-Called-The Library-System.
...
2> (Define University-Library-Database :System :Synonym Uldb)
...
3> (Define University-Library :Environment :Library :Synonym Ul)
Ul Is-An-Instance-Of "Library".
...
4> (Need (!Logic (Tracks Uldb Ul)))
Uldb Is-An-Instance-Of Tracking-System.
...
5> (Show-Pending-Issues)
1- Need-Definition of "Library".
2- Need-Further-Disambiguation of Tracks.
3- Need-Definition (Item-States Ul).
6> (Define Library :Ako Repository :Defaults (:Collection-Type Book))
Ul.Collection-Type Has-Value "Book".
...
7> (Define Book :Ako Physical-Object :Member-Roles (Title Author Isbn))
...
8> (Define Book.Isbn :Ako Integer :Cardinality Single :Unique-Id T)
...
9> (Define Unshelf :Roles (:Records Remove))
Uldb Is-An-Instance-Of Tracking-Information-System.
Uldb.Manner-Of-Observation Is-An-Instance-Of Indirect-Observation.
Unshelf.Object-Type Has-Value Book.
Unshelf.Records Has-Value Remove-Repository.
...
10> (Define Unshelf-All :Roles
      (:Records Remove
       :Objects (!The-Set-Of-All (?B) Such-That (= (Isbn ?B) $Input))))
...
Conflict #1 Colliding-Definitions-Of Unshelf And Unshelf-All
11> (Reformulate Book Copy-Of-Book)
<6>(Define Library :Ako Repository :Defaults (:Collection-Type Book))
=Reformulate=>?(Y or N) Yes.
<7> (Define Book :Ako Physical-Object :Member-Roles (Title Author Isbn))
=Reformulate=>?(Y or N) Yes.
<8> (Define Book.Isbn :Ako Integer :Cardinality Single :Unique-Id T)
=Reformulate=>?(Y or N) No.
...
12> (Define Books-By-Topic :Report :Roles
      (:Data-In (!Decl ((?Topic Topic)))
       :Data-Out (!Decl ((?Titles (!Set-Of Title))))
       :Query (!Logic (= ?Titles (!The-Set-Of-All (?T Title) Such-That
                                    (!There-Exists (?B Book) Such-That
                                      (And (Mem ?B Ul)
                                           (= ?T ?B.Title)
                                           (= ?Topic ?B.Topic)))))))))
Book Has-A :Property Called "Topic".
...
```

Figure 4: Interaction between an analyst and the Requirements Apprentice, excerpted from a much longer transcript in [4], which illustrates the full capabilities of the Apprentice. Programmer input follows the > prompts.

1 Introduction

The environment of the library-system requirement is the University-Library (UL). The UL is a library. A library is a repository for books.

The system being specified is the University-Library-Database (ULDB). The ULDB is a tracking-information system, which tracks the state of the (UL). Three transactions and one report are specified.

The library-system requirement is incomplete. There are five pending issues.

4.2 Reports of ULDB

The reports of an information-system generate information about the database without altering it. Reports are principally described by an input data signature, an output data signature, and a query that defines the functionality of the report.

4.2.1 Books-By-Borrower

The tracking-information-system-report books-by-borrower provides information about what the state of the UL is believed to be.

Data-in: ?topic of type topic.
Data-out: ?titles of type set-of title.
Query: ?titles = {?t:title | ∃ ?b:book (?b ∈ UL ∧ ?t = ?b.title ∧ ?topic = ?b.topic)}.
Accessed-information: titles of books and topics of books.
Target: UL.

The purpose slot is empty.

Figure 5: Two excerpts from a requirements document generated at the end of Figure 4.

to assume that this knowledge will ever be complete. In consonance with this, the RA's current cliché library contains extensive knowledge of information systems, tracking systems, and repositories but no knowledge about libraries or library information systems *per se*.

The fourth command in Figure 4 states the key requirement that the ULDB system tracks the library UL. As illustrated by the output generated by the fifth command, much remains vague at this point. In particular, the term library needs to be defined, and the term tracks is ambiguous because it is not yet clear which kind of tracking system is intended. Since incompleteness and ambiguity are inevitable during the early stages of constructing a requirement, the Apprentice refrains from complaining at this point. Rather, it accepts information and performs inferences on a catch-as-catch-can basis.

After giving brief definitions of the terms *library* and *book*, the analyst uses the ninth command to begin the process of defining the functional requirements. The processing triggered by this command is an interesting example of the way the RA operates. The command is ambiguous because, while the words *records* and *remove* are both mentioned in the cliché library, neither one has a unique definition. This problem is resolved by locating something that can *record remove* and is meaningful in the context of a tracking system. In particular, the ULDB is further specialized to a tracking information system.

This allows *unshelf* to be understood as a tracking operation (see Figure 3). In addition, since remove corresponds to one of the fundamental operations that alter the state of a repository, it can be concluded that the ULDB operates by indirect-observation. All this information is recorded in the RKB for future reference.

The first command to trigger a major response from the RA is the analyst's attempt to define an *unshelf-all* transaction that records the removal of every volume with a given ISBN number. The RA complains that, as currently defined, unshelf is identical to unshelf-all. The detection of this problem is supported by Cake based on equality reasoning and simple deduction. (Since ISBN numbers are unique identifiers for books, there cannot be more than one book with a given ISBN.) The RA complains about this because it has a built-in bias that new terms should not be synonymous with old terms unless explicitly declared to be so.

On seeing this conflict, the analyst realizes that the commands to this point reflect a type/token confusion between a physical copy of a book and the logical notion of a book as a class of copies of books with some particular title, author, and ISBN.

To fix the problem, a new word corresponding to a copy of a book must be introduced. In addition, something has to be done about the fact that only some uses of the word book in the commands above refer to the concept of a book. The rest refer to the concept of a copy of a book. The RA provides a special command *reformulate* (see Figure 4) that can assist with this kind of conceptual realignment. Using the dependency information maintained by Cake, the RA propagates the conceptual reorganization throughout the RKB.

With the last command in Figure 4, the analyst defines a report that the library management system needs to support. As noted in conjunction with Figure 2, the principle output of the RA is the RKB. However, the RA is capable of creating a requirements document at any time. Portions of the document created after the twelfth command in Figure 4 are shown in Figure 5.

The transcript in Figure 4 shows that the RA is able to give powerful support to an analyst during the process of creating a formal requirement based on an informal requirements sketch. A vital underpinning of this is a deep representation of the evolving requirement that allows the RA to reason about it. However, the most important single factor is the library of requirements clichés. This enables the analyst to rapidly construct a requirement *by inspection*, by combining requirements clichés. Given the complexities of requirements analysis, it is natural for the RA to function as an assistant to a requirements analyst rather than attempting to automate the entire requirements analysis process.

Other Examples

To see the breadth of applicability of deep representations, inspection methods, and intelligent assistance, it is useful to consider how they can be applied to areas other than requirements.

Program design and implementation. A variety of current tools assist with program construction, including editors, cross-referencers, program generators, and component libraries. In general, these tools operate at the level of source code, sometimes

augmented with parse trees, compiler symbol tables, and the like.

To go dramatically beyond the level of current tools and support complex algorithmic changes in a program, one has to move beyond superficial representations like parse trees and use direct representations of control and data flow such as the Plan Calculus [6] to represent programs and their designs. As compared to program text or parse trees, the Plan Calculus makes data flow, control flow, and the design of a program more explicit. This facilitates the direct manipulation of these features. For example, cross-referencing could be more accurate: When searching for where a variable is set, one could ignore appearances of the variable that were not in the relevant control-flow path.

Program construction tools could also benefit from the introduction of inspection methods, as illustrated by the Knowledge-Based Editor in Emacs [6]. This system demonstrates that a library of implementation clichés represented as plans makes it possible to support the rapid construction and modification of programs by means of commands that are phrased in terms of algorithms and their structure rather than program text. An important feature of this is that using the Plan Calculus as a representation for algorithmic clichés supports more abstract and canonical component libraries, as compared to using subroutines or program templates.

Looking further into the future, it will be possible to deepen the representations used in program construction in the direction of making design decisions explicit, as illustrated in the Design Apprentice scenario in [8].

Reverse engineering. The term *reverse engineering* has come into use recently to describe the application of software engineering tools to existing software, i.e., software not necessarily produced using advanced tools in the first place. Approaching this problem using shallow representation and procedural methods has led to the development of a number of useful capabilities.

For example, program restructuring tools are now generally available for a number of languages. These tools, operating mostly on the level of program syntax, improve the understandability of source code by replacing go-to instructions with if-then-else, do-while, and other structured programming constructs. Other reverse engineering tools help a programmer understand existing software by displaying the static or dynamic structure of the code graphically and allowing the programmer to easily navigate within it.

The power of these tools to improve understandability could be increased by using an explicit representation of algorithmic structure, such as the Plan Calculus. For example, given a complete data- and control-flow analysis, it would be possible to eliminate not only dead (unexecutable) code, but also code that was executed but whose results were not used. Needless shuffling of values among intermediate variables could similarly be eliminated. It is likely that the internal operation of restructuring tools would also be simplified by using such a representation, as compared to operating directly on the source code.

A much deeper level of reverse engineering is illustrated by the Recognizer implemented by Wills [7, 9]. Based on a library of design and implementation clichés represented using the Plan Calculus, the Recognizer takes a program (which is also represented using plans) and parses it to determine how the program could have been constructed using the clichés. Once an on-line representation of its design has been obtained, the

program can then be modified by other tools at the level of design decisions, just as if it had originally been developed using a design-level tool.

Totally automatic design reconstruction is not likely to be practical for programs of realistic size. Therefore, an intelligent assistance approach will be necessary. The software engineer will need to be involved in the process both to reduce the large search required and to provide specification information, such as the expected range of input values, that is missing from the source code.

Testing. Software testing comprises two major subtasks: devising test cases and executing them. Deeper representations (such as data and control flow) and inspection methods can help in both of these areas.

Current testing tools already make use of control-flow representations to check the *coverage* of a set of test cases, i.e., whether each path through a program is executed by some test case. Unfortunately, even when using data-flow representations, it is in general not possible to automatically construct test cases with complete coverage. Inspection methods provide a complementary approach. There are testing clichés, just as there are requirements, design, and implementation clichés. For example, it is well known that for buffered input/output routines it is a good idea to test what happens when the buffer gets full. An intelligent assistant for testing could suggest clichéd test cases based on the design of a program.

The major difficulty in executing test cases is deciding which ones to run after a design change when it is not feasible to run them all. A deeper representation would help here also: If test cases are indexed according to the design features they test, rather than just the parts of the code they exercise, it would be easier to select the most relevant tests. A modest step in this direction is demonstrated in [1].

Systems analysis. At the heart of all current systems analysis tools is some kind of diagram editor. Engineers can interactively draw boxes and arrows on the screen, with various annotations on them indicating the type of data, type of operation, etc. The editor checks that the diagrams are well formed with respect to the syntactic rules of one of the standard diagramming methodologies.

The information content of the hierarchical box-and-arrow diagrams used in systems analysis tools is typically represented at a fairly deep level. However, these tools could benefit from more emphasis on clichés and inspection methods.

The next generation of diagram editors for systems analysis could improve on current diagram editors in much the same way that the RA improves on current requirements tools. In current diagram editors, the engineer constructs a system description either from scratch or by cutting and pasting from existing diagrams, both of which can be tedious and error-prone. A library of systems analysis clichés would be more than just a collection of already constructed diagrams. Each systems analysis cliché would be a schematized diagram with associated defaults, constraints, and explanations. The library of clichés would be organized taxonomically. As in the RA, the editor would keep track of how a diagram was constructed out of clichés and support later modification in the same terms.

It may also be useful to automatically recognize occurrences of clichés in existing diagrams, in the same way the Recognizer previously described recognizes occurrences

of clichés in plans. The knowledge associated with clichés in the library may then be applied to existing diagrams.

Looking further into the future, it will be possible to deepen the representations used in systems analysis in the direction of more general logical reasoning, which could be applied, for example, to automating trade-off analysis and checking for deeper logical inconsistency and incompleteness as in the RA.

An energy barrier. As illustrated by the examples above, deep representation, inspection methods, and intelligent assistance have the potential of supporting a new generation of knowledge-intensive software engineering tools significantly more powerful than those that can be supported with current technology. The current emphasis on simpler approaches is probably due to the fact that knowledge-intensive approaches have a significantly larger minimum cost (see Figure 1). This cost forms an energy barrier which quite naturally inhibits the use of these approaches as long as significant progress can be made with shallow representations and procedural methods. However, we believe that the potential of these simple approaches is nearing exhaustion and that knowledge-intensive techniques will come to the fore over the coming decade.

References

[1] D. Chapman, A Program Testing Assistant, *Comm. of the ACM*, 25(9):625–634, September 1982.

[2] E. J. Chikofsky, *Computer-Aided Software Engineering (CASE)*, IEEE Computer Society Press, Los Alamitos, CA, 1989.

[3] C. Gane, *Computer-Aided Software Engineering: The Methodologies, the Products, and the Future*, Prentice Hall, Englewood Cliffs, NJ, 1990.

[4] H.B. Reubenstein and R.C. Waters, "The Requirements Apprentice: Automated Assistance for Requirements Acquisition", *IEEE Transactions on Software Engineering*, 17(3):226–240, March 1991.

[5] C. Rich, "The Layered Architecture of a System for Reasoning about Programs", *Proc. of the 9th Int. Joint Conference on Artificial Intelligence*, pp. 540–546, August 1985.

[6] C. Rich and R.C. Waters, *The Programmer's Apprentice*, Addison–Wesley, Reading MA and ACM Press, Baltimore MD, 1990.

[7] C. Rich and L. M. Wills, "Recognizing a Program's Design: A Graph-Parsing Approach", *IEEE Software*, 7(1):82–89, January 1990.

[8] R.C. Waters and Y.M. Tan, "Toward a Design Apprentice: Supporting Reuse and Evolution in Software Design", *ACM SIGSOFT Software Engineering Notes*, 16(2):33–44, April 1991.

[9] L. M. Wills, "Automated Program Recognition: A Feasibility Demonstration", *Artificial Intelligence*, 45(1-2):113–171, September 1990.

[10] *Proceedings of the 3rd, 4th, and 5th Int. Workshops on Software Specification and Design*, Computer Society Press, Washington DC, 1985, 1987, and 1989.