Approaches to Automatic Programming

Charles Rich, Richard C. Waters

TR92-04 July 1992

Abstract

This paper is an overview of current approaches to automatic programming organized around three fundamental questions that must be addressed in the design of any automatic programming system: What does the user see? How does the system work? What does the system know? As an example of a research effort in this area, we focus the Programmerś Apprentice project.

Advances in Computers, Volume 37, M. C. Yovits, ed., Academic Press, 1993

Publication History:–

1. First printing, MN92-04, July 1992

1 Introduction

Automatic programming has been a goal of computer science and artificial intelligence since the first programmer came face to face with the difficulties of programming. As befits such a long-term goal, it has been a moving target—constantly shifting to reflect increasing expectations.

Much of what was originally conceived of as automatic programming was achieved long ago. Today, no one would call an assembler or a compiler automatic programming. However, when these devices were first invented in the 1950's, the term was quite appropriate. Compared with programming in machine code, assemblers represented a spectacular level of automation. Moreover, Fortran was arguably a greater step forward than anything that has happened since. In particular, it dramatically increased the number of scientific end users who could use computers without having to hire a programmer.

On the other hand, current expectations regarding the potential of automatic programming are often based on an idealized view of reality and are probably unachievable. Nevertheless, a number of important developments are appearing in research efforts and commercially available systems. As an example of a research effort in this area, we focus on our project at MIT, called the Programmer's Apprentice (Section 5).

The "cocktail party" description of the potential of automatic programming runs something like this:

> There will be no more programming. The end user, who only needs to know about the application domain, will write a brief requirement for what is wanted. The automatic programming system, which only needs to know about programming, will produce an efficient program satisfying the requirement.
>
> Automatic programming systems will have three key features: They will be *end-user oriented*, communicating directly with end users; they will be *general purpose*, working as well in one domain as in another; and they will be *fully automatic*, requiring no human assistance.

Although this description is attractive, it is based on a number of faulty assumptions.

1.1 Myth: End-user-oriented automatic programming systems do not need domain knowledge

It is no more possible for end users to communicate effectively with an automatic programming system that knows nothing about the application domain than it is to communicate effectively with a human programmer who knows nothing about the application domain. Rather, the path from an end-user's needs to a program involves a gradual change from a description that can only be understood in the context of the domain to a description that can be understood without relying on auxiliary knowledge. There is no point at which someone who knows nothing about programming communicates directly with someone who knows nothing about the application domain.

For example, suppose a large company needs a new accounting system. Figure 1 shows the principal agents that would typically be involved. The bars on the right indicate that near the top of the figure accounting knowledge plays the crucial role, while programming knowledge dominates towards the bottom.

The manager at the top of the figure quite likely has only a rudimentary knowledge of accounting. The manager's job is to identify a need and initiate the programming process by creating a brief, vague requirement. The term "vague" is used here to highlight the fact that the only way this initial requirement can succeed in being brief is for it to also be incomplete, ambiguous, and/or inconsistent.

The next agent in the process is an accounting expert. The accounting expert's job is to take the manager's vague requirement and create a detailed requirement. A key feature of this requirement is that it is couched in the technical vocabulary of accounting and is intended to be evaluated by other accounting experts. The accounting expert's knowledge of programming does not have to extend much beyond basic notions of feasibility.

The third agent in the process is some sort of system analyst. The analyst's job is to define the basic architecture of the program and to translate the requirement into a detailed specification. In contrast to the requirement, the specification is couched in the technical vocabulary of programming, rather than accounting. To perform this transformation, the system analyst must have a considerable understanding of accounting in addition to an extensive knowledge of programming. (Section 5.1 demonstrates an interactive tool to assist systems analysts.)

The final agent in the process is a programmer. The programmer's job is to produce code in a high level language based on the detailed specification. The programmer does not have to know very much about accounting. However, it is very unlikely that the accounting system will actually work if the programmer knows nothing about accounting.

Although not shown in the figure, agents for validation, testing, documentation, and modification are required as well. To do their jobs, these agents also need significant domain knowledge.

In reality, end-user-oriented automatic programming systems must be domain experts. In particular, it is not possible to describe something briefly unless the hearer knows at least as much about the domain as you do and can therefore understand the words used and fill in what is left out. (For a further discussion of the necessity of domain knowledge in automatic programming, see Barstow (1984).)

1.2 Myth: End-user-oriented, general-purpose, fully automatic programming is possible

A corollary of the need for domain knowledge is that such an automatic programming system would have to be an expert in every application domain. Unfortunately,

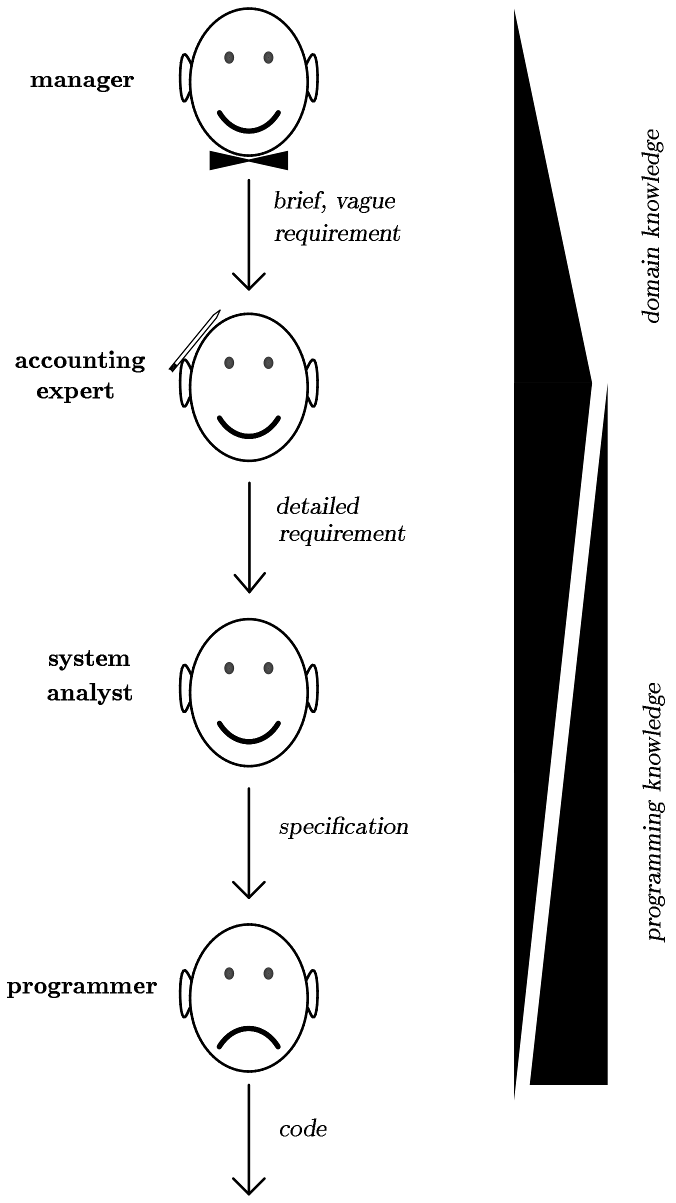

Figure 1. Agents in an accounting software project and the kind of knowledge they use.

there is no hint that artificial intelligence is anywhere near supporting this kind of superhuman performance.

Given the pragmatic impossibility of simultaneously supporting all three features mentioned above, it is not surprising that all current approaches to automatic programming focus on two of the features at the expense of the third. This has given rise to the following three approaches to automatic programming, typified by the feature given up:

a. *Bottom-up.* This approach sacrifices end-user orientation. It starts at the programmer's level and tries to push the threshold of automation upward. In the past, the threshold was raised from machine-level to high-level languages. The current goal is to raise the threshold further to so-called very high-level languages.

b. *Narrow-domain.* This approach sacrifices being general purpose. Focussing on a narrow enough domain makes it feasible right now to construct a fully automatic program generator that communicates directly with end users. This approach is advancing to cover wider domains.

c. *Assistant.* This approach sacrifices full automation. Instead, it seeks to assist in various aspect of programming. (The Programmer's Apprentice pioneered the application of artificial intelligence technology to the assistant approach.) With current technology, this approach is represented by programming environments consisting of collections of tools such as intelligent editors, on-line documentation aids, and program analyzers. The goal here is to improve the integration between tools and level of assistance provided by individual tools.

1.3 Myth: Requirements can be complete

Since the cocktail party description of automatic programming assumes that the only point of contact between the end user and the system is a requirement, this requirement must be complete. In the interest of producing an efficient program, the automatic programming system is expected to take full advantage of every degree of freedom allowed by the requirement. The completeness of the requirement guarantees that anything the automatic programming system produces will be acceptable to the end user.

This point of view is commonly justified by likening requirements to legal contracts. However, any lawyer will tell you that contracts do not work that way. Contracts only work when both parties make a good faith effort to work toward a common end. If good faith breaks down, the parties can always cheat without violating the "letter" of the contract.

The problem with requirements (and contracts) is that they *cannot* be complete. No matter how trivial the situation in question, there is no practical limit to what needs to be said when trying to pin down a potential adversary.

Consider, for example, specifying a controller for an automated teller machine. When describing the withdrawal operation, it is easy enough to say that after the bank card is inserted, the customer should enter a password, specify an account, and then select an amount of cash which the machine then dispenses. However, this is nowhere near complete.

To start with, there are a lot of details missing with regard to the user interface: What kinds of directions are displayed to the customer? How is the customer to select between various accounts? What kind of acknowledgment is produced? To be complete, these details must include the layout of every screen and printout, or at least, a set of criteria for judging the acceptability of these layouts.

Even after the interface details are all specified, the requirement is still far from complete. For example, consider just the operation of checking the customer's password. What are passwords to be compared against? If this involves a central repository of password information, how is this to be protected against potential fraud within the bank? What kind of response time is required? Is anything to be done to deal with possible tampering with bank cards?

One step deeper, a truly complete requirement would have to list *every* possible error that could occur—in the customer's input, the teller machine, the central bank computer, the communication lines—and state exactly how each error should be handled.

Going beyond what is computed, the user undoubtedly wants the program to be reasonably efficient. This could be specified as maximum limits on space and time. However, what is really desired is for the implementor to make a good faith effort to make the program as efficient as possible. Further, the code produced should be easy to read, easy to modify, and well documented.

Finally, the end user also undoubtedly cares about the cost of implementing the program and how long it will take. This implies that trade-offs must be made, particularly when it comes to the last few issues mentioned above. This makes it very difficult—if not impossible—to make complete statements about these issues.

In reality, requirements are at best only approximations. Instead of serving as a defensive measure between adversaries, requirements should be used as a tool for communication between colleagues. Assuming that the implementor will make a good faith effort to create a reasonable program, many of the points above can go unsaid.

Just like human programmers, an automatic programming system must make a good-faith effort to satisfy the spirit of the requirements given to it. The system must be oriented toward making reasonable assumptions about unspecified properties, rather than trying to minimally satisfy specified properties. This observation reinforces the need for domain knowledge as part of an automatic programming system.

1.4 Myth: Programming is a serial process

In many ways, the worst aspect of the cocktail party description of automatic programming is that it perpetuates the myth that creating a program is a two step process: First, an end user creates a requirement; second, the automatic programming system makes a program. This view is just as impractical in the context of an automatic programming system as it is in human-based programming.

First of all, given the approximate nature of requirements, a considerable amount of back-and-forth communication is required to convey the end user's full intent. Second, users typically start the programming process with only a vague idea of what they want, and they need a significant amount of feedback to flesh out their ideas and determine the desired requirement. Also, what end users want today is never the same as what they want tomorrow. Third, users do not want programmers to follow requirements blindly. If problems arise, they want advice. For example, the programmer should tell the user if a slight relaxation in the requirement would allow a much more efficient algorithm to be used.

In reality, programming is an iterative process featuring continual dialogue between end user and programmer. The desired requirement evolves out of prototypes and initial versions of the system.

The inherently iterative nature of programming has two important implications for automatic programming. First, just as in nonautomatic programming, the focus of activity will be on *changing* requirements as much as on implementing them. Thus, there will be no reduction in the need for regression testing and other techniques for managing evolution.

Second, to carry on a dialogue with the user, automatic programming systems will need to explain what they have done and why. In particular, they will need to explain the assumptions they have introduced into a requirement so that users can debug those assumptions.

1.5 Myth: There will be no more programming

There will certainly be many differences between the input to future automatic programming systems and what is currently called a program. However, programming is best typified not by what programs are like but by what programming *tasks* are like. In particular, these new inputs will undoubtedly still have to be carefully crafted, debugged, and maintained according to changing needs. Whether or not one chooses to call these inputs programs, the tasks associated with them will be strongly reminiscent of programming.

In reality, end users will become programmers. As an example of this phenomenon, consider spreadsheet programs. When spreadsheets first appeared, they were heralded as a way to let users get their work done without having to deal with programmers or learn programming. Spreadsheets have succeeded admirably in letting users get results by themselves. However, maintaining a complex spreadsheet over time differs very little from maintaining a program. The only real difference is that a spreadsheet is a concise domain-specific interface which makes it remarkably easy to write certain kinds of programs and startlingly difficult to write other kinds of programs.

1.6 Myth: There will be no more programming in the large

Even if we accept the fact that programming will be around forever, we might well hope that by continuing the trend of writing programs more compactly, automatic programming will convert all programming into programming in the small.

Unfortunately, this dream overlooks software's extreme elasticity of demand. Most of the productivity improvements introduced by automatic programming will almost certainly be utilized to attack applications that are enormous rather than merely huge.

In reality, we are unlikely to ever settle for only those application systems that can be created by a few people. As a result, there will be no lessening of the need for version control, management aids, and all of the other accoutrements of cooperative work and programming in the large.

The automatic programming systems of the future will be more like vacuum cleaners than like self-cleaning ovens. With a self-cleaning oven, all you have to do is decide that you want the oven cleaned and push a button. With vacuum cleaners, your productivity is greatly enhanced, but you still have a lot of work to do.

The next three sections discuss the three fundamental technical issues in automatic programming that must be addressed in the design of any automatic programming system: What does the user see? How does the system work? What does the system know?

2 What Does the User See?

From the user's perspective, the most prominent aspect of an automatic programming system is the language used to communicate with it. We will discuss the range of possibilities below using as an example the simple problem of determining the value of an octal number represented as a string.

Figure 2 shows the Pascal implementation of a program that solves this problem. We would like an automatic programming system to produce a program like this from one of the user inputs discussed below. Note that no single example can equally well illustrate all of the important issues in selecting an input medium.

2.1 Natural Language

Because they are familiar, natural languages such as English are an attractive choice for communication between end users and an automatic programming system. Three other features that make natural language attractive are vocabulary, informality, and

```
function EvalOctal (var S array [M..N: Integer] of Char): Integer;
  var I, V: Integer;
begin
  I := M;
  V := -1;
  while S[I]=' ' do I := I+1;
  if S[I]<>chr(0) then begin
     V := 0;
     while ('0'<=S[I]) and (S[I]<='7') do begin
       V := 8*V+ord(S[I])-ord('0');
       I := I+1
       end;
     while S[I]=' ' do I := I+1;
     if S[I]<>chr(0) then V := -1
     end;
  EvalOctal := V
end
```

Figure 2. Pascal implementation of a program that determines the value of an octal number represented as a string. Note that to allow for strings of zero length, strings are implemented as arrays of characters where the last character is not part of the string. The function `ord` returns the integer character code corresponding to a character.

syntax. For example, the following is a natural language specification of the `EvalOctal` program in Figure 2.

> The function `EvalOctal` is a recognizer that determines whether or not a given string contains an octal number optionally surrounded by blanks. If this is the case, the decimal value of the number is returned; otherwise `-1` is returned.

The feature that contributes the most to making natural language an efficient communication medium is the existence of a vocabulary of thousands of predefined words. Among other things, the natural language specification above assumes that the reader knows what a recognizer is, what an octal number is, and the relationship between the value and string representation of an octal number.

Informality, such as the possibility of a statement's being ambiguous, incomplete, contradictory, and/or inaccurate, is also very important. In fact, it essential to a powerful strategy for dealing with complexity: Start with an almost-right description and incrementally modify it until it is acceptable.

For example, the natural language specification above is vague about the data type of the output. (`Integer` was chosen in Figure 2.) It is also vague about whether or not the octal number may be preceded by a sign. (Figure 2 assumes it may not, since if negative numbers were allowed, then `-1` could correspond to a valid input.) Finally, the specification above does not say what should happen if the octal number is too big to represented. (Figure 2 allows overflow to occur.)

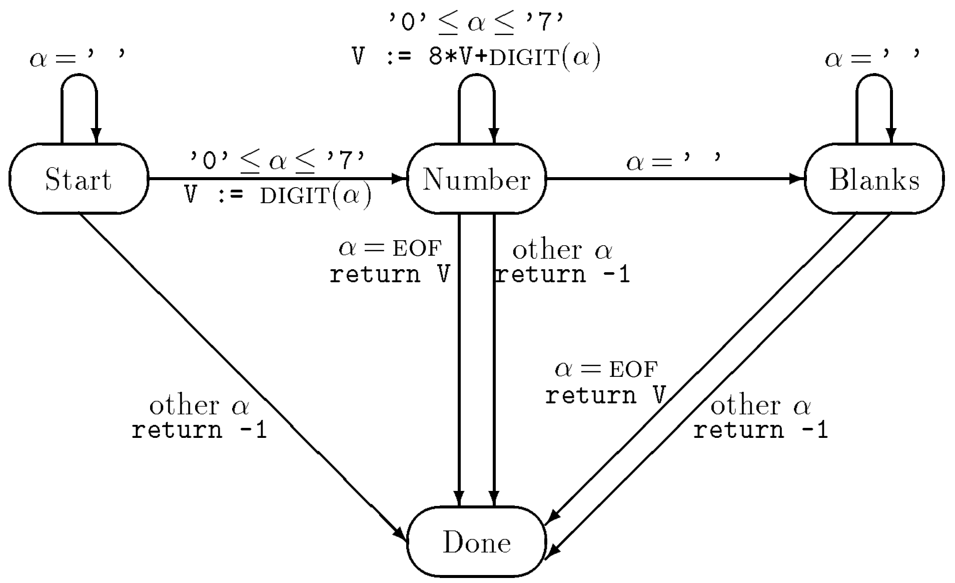

Figure 3. A state transition diagram specification for `EvalOctal`.

The least important feature of natural language is its syntax. Natural syntax is convenient, because it is familiar. However, it is of relatively little value unless the other features are supported as well.

Unfortunately, enabling machines to converse in natural language is far beyond the current abilities of artificial intelligence. As a result, natural language input—although an active area of inquiry in its own right—is not a major topic in current automatic programming research. For examples of early work on automatic programming using natural language input, see Heidorn (1976) and Ruth (1978).

2.2 Special-Purpose Languages

Even when people communicate among themselves, natural language is not always the language of choice. For example, many application areas have specialized symbolic or graphical languages associated with them, such as mathematical formulae and circuit diagrams, that experts routinely use in preference to natural language.

For example, Figure 3 shows the specification of `EvalOctal` using the standard graphical notation for describing state transition networks. In this diagram, states are represented as ovals and transitions between states are represented as arrows. The arrows specify what happens when an input character α is encountered. (For instance, suppose the program is in the Blanks state. If it receives a blank character, then it stays in the Blanks state. If it receives an EOF character indicating that there is no more input, then it goes to the Done state.) In addition, some of the arrows are annotated with specifications for actions to be performed when transitions occur. (For instance, if the program is in the Start state and receives one of the characters `'0'` through `'7'`, it sets a variable `V` equal to the integer corresponding to the character.)

A key advantage of Figure 3 is that it is easy to understand and modify. Specification changes can be easily made using, for example, a graphical editor.

Figure 3 is also much more formal and detailed than the natural language description of `EvalOctal` in Section 2.1. For example, it specifies exactly what can appear in the input. However, it is also still incomplete. For example, it does not say what to do if the input number is too big.

Many kinds of special-purpose languages can be supported in straightforward ways, as long as their focus is sufficiently narrow. A particularly successful example is so-called "what you see is what you get" interfaces. Screen painters allow end users to specify the layout (and some of the semantics) of a data entry and retrieval program simply by making a picture of how the screen should look. Then a code generator automatically writes the code to drive the terminal and access the data base.

Unfortunately, special-purpose languages have the fundamental problem that they are essentially useless outside of their domains of applicability. This brings up a key unsolved problem—namely, how to combine several special-purpose languages or a special-purpose language with a general-purpose one.

Almost every current system that supports a special-purpose language follows the narrow-domain approach to automatic programming, restricting itself to the situations where the special-purpose language is appropriate. Even when multiple, special-purpose input languages are supported (e.g., in Draco (Neighbors, 1984)), the user is only allowed to combine the languages in simple ways. Much more work needs to be done before special-purpose languages can reach their full potential as part of the interface to general-purpose systems.

2.3 Examples

An attractive idea, pursued with some vigor in the early days of automatic programming, is to specify a program via examples of its behavior. Early work in this area (Hardy, 1975; Shaw et al., 1975; Siklossy and Sykes, 1975) was *ad hoc* in nature. This changed with the work of Summers (1977) who established theoretical foundations for the field. Most subsequent work is based on Summers' approach. Furthermore, most of the work on programming by example uses Lisp as the target language (see survey by Smith (1984)). The reason for this is not intrinsic to any of the techniques, but rather it is because the Lisp programming environment facilitates writing programs that operate on programs.

The appeal of programming by example is that non-programmers are familiar with examples as a communication technique, just as they are with natural and special-purpose languages. Furthermore, collections of examples are easy to understand and modify. For example, the following is a set of input-output examples which might be used to specify `EvalOctal`.

```
"132 " ⇒ 90
" 42 " ⇒ 34
"  56" ⇒ 46
"-17 " ⇒ -1
"2.6 " ⇒ -1
" 380" ⇒ -1
```

Individually, these examples seem clear enough. However, if this truly was the only information given, it would be next to impossible for a person (let alone an automatic system) to understand what the program was supposed to do.

To start with, considerable insight is required to come up with the correct generalization of the first three examples—i.e., that the output is the value of the input viewed as an octal number. In addition, there need to be many more examples to address questions like: What happens if there are more than three digits? What happens if there is more than one blank after the number? What happens if there is a `'+'` in front of a number. What happens if a number starts with a `'0'`? What happens if two numbers are separated by blanks? What happens if there are letters in the input? (Imagine how many examples would be required if `EvalOctal` were not a toy program!)

Although it initially created a good deal of excitement, programming by example now appears to be of only theoretical interest. The problem is that the techniques do not scale up to programs of realistic complexity. Unfortunately, as input-output pairs become more complex, the problem of generalizing them appropriately becomes astronomically more complex.

There are basically only two ways to cut the generalization problem down to a manageable size. First, more examples can be provided in order to reduce ambiguity, including examples of intermediate computation steps (see Biermann (1972), Biermann (1976), Biermann and Krishnaswamy (1976), and Bauer (1979)). Unfortunately, when the number of examples becomes too large, this becomes an inefficient means of specification as compared with other formal techniques.

Alternatively, assumptions about the class of target programs can be built into the programming-by-example system. For example, Andreae (1985) synthesizes robot programs from examples. The system described by Hedrick (1976) assumes that the synthesized program must be a rule-based production system. Unfortunately, when the structural assumptions begin to get this strong, the programming-by-example system ends up being essentially a special kind of program generator.

A different approach to using examples is illustrated by the Tinker system (Lieberman and Hewitt, 1980). Tinker does not attempt to generalize the examples automatically but rather provides a program development environment that helps the user perform the generalization. In this context, the input-output examples are perhaps better thought of as test cases.

In summary, examples may be a useful technique only as an adjunct to other specifications or in a narrow domain in which the possible programs are already highly restricted.

2.4 Logical Formalisms

Logic is the most powerful (and general) formal description language known. As a result, it is reasonable to suppose that it might make a good communication medium between a user and an automatic programming system.

Unfortunately, there are two fundamental barriers to the use of logical formalisms. First, most interesting tasks in general logical systems (for example, detecting contradictions) are computationally intractable (see the discussion of deductive methods in Section 3.2). Second, complex logical formulae are notoriously difficult for most people to write and understand.

For example, the following is a logical specification for `EvalOctal` in the form of pre- and postconditions. This specification assumes that the notion of a string is built into the logic. In particular, a string S is treated as a function that maps from the integer range $1..|S|$ to characters, where $|S|$ denotes the length of a string.

input: S
precondition: $\textsc{string}(S)$
output: V
postcondition: $\textsc{integer}(V) \wedge (\textsc{valid}(S) \rightarrow \textsc{value}(S,V)) \wedge (\neg\textsc{valid}(S) \rightarrow V = -1)$

where:

$$\begin{aligned}
\textsc{valid}(S) \;&\equiv\; (\forall i\; 1 \le i \le |S| \rightarrow S(i) \in \{\texttt{' '}, \texttt{'0'}, \texttt{'1'}, \texttt{'2'}, \texttt{'3'}, \texttt{'4'}, \texttt{'5'}, \texttt{'6'}, \texttt{'7'}\}) \\
&\quad \wedge\; (\exists i\; 1 \le i \le |S| \rightarrow S(i) \ne \texttt{' '}) \\
&\quad \wedge\; (\neg\exists ijk\;\; 1 \le i < j < k \le |S| \wedge S(i) \ne \texttt{' '} \wedge S(j) = \texttt{' '} \wedge S(k) \ne \texttt{' '}) \\
\textsc{value}(S,V) \;&\equiv\; \forall ij\; (1 \le i < j \le |S| \wedge S(i) \ne \texttt{' '} \wedge S(j) \ne \texttt{' '} \wedge (j = |S| \vee S(j{+}1) = \texttt{' '})) \\
&\qquad\quad \rightarrow\; (\textsc{digit}(S(i)) = \textsc{div}(\textsc{rem}(V, 8^{j-i-1}), 8^{j-i-2}) \\
&\qquad\qquad\quad \wedge\; ((i = 1 \vee S(i{-}1) = \texttt{' '}) \rightarrow V < 8^{j-i}))
\end{aligned}$$

The essence of this specification is contained in the definitions of VALID and VALUE. The VALID predicate is true of a string iff every character is an octal digit or a blank, at least one character is a digit, and there are no blank characters between digits. The VALUE relation holds true iff each digit in S is correctly represented in V and V contains no other values. (The DIGIT function returns the integer corresponding to a digit character. DIV and REM are the integer division and remainder operators, respectively.)

The specification above has the virtue of being very precise without biasing possible implementations. Unfortunately, it is far from obvious, for example, that the VALUE relation in fact guarantees that V is the value of the octal number represented in S.

Research on logic as a communication medium between man and machine is being carried out primarily under the topics of formal specification languages and logic-programming languages. A key issue in both of these areas is the introduction of extensions and restrictions that render logic more tractable to man and machine. For example, Prolog (Cohen, 1985) guarantees executability of logical descriptions by placing strong restrictions on the form of expressions.

2.5 Very High Level Languages

While specification languages and logic programming languages essentially extend "downward" from logic, very high level languages build "upward" from current high level languages. Typically, very high level languages add powerful abstract datatypes, such as sets and mappings (to allow programmers to ignore the details of data structure implementation), and a few features of logical notation, such as quantification over sets (to allow programmers to ignore certain kinds of algorithmic detail).

The archetype of very high level languages is SETL (Schwartz et al., 1986). SETL supports most of the standard constructs of Algol-like programming languages. In addition, it supports two convenient universal data structures—tuples and sets. For example, a mapping is treated as a set of 2-tuples. SETL also supports the use of universal and existential quantifiers in a program. For example, the following is the form of the SETL statement for performing some computation on every element of the set `S`.

```
(forall x in S) ... end forall;
```

One of the main goals of providing such expressive facilities is to free the programmer from having to think about the detailed design of data structures. The SETL compiler will decide how data structures should be implemented. This decrease in what the programmer has to worry about is a key to the productivity gains that should be obtained by the use of very high level languages.

The following shows how one could write `EvalOctal` in SETL.

```
procedure EvalOctal(S);
  if (forall C in S | C in {' ','0','1','2','3','4','5','6','7'}) and
     (exists C in S | C /= ' ') and
     (not exists Ci in S(i), Cj in S(j), Ck in S(k)
       | i<j and j<k and Ci/=' ' and Cj=' ' and Ck/=' ')
    then Digits := [abs C - abs '0': C in S | C/=' '];
         return +/[D*8**(#Digits-i): D in Digits(i)];
    else return -1;
  end if;
end procedure EvalOctal;
```

The form of this program closely follows the form of the logical specification in the preceding section. The condition of the `if` statement illustrates that many logical expressions can be rendered more or less directly in SETL. However, this cannot be done with non-constructive logical expressions such as VALUE relation.

When the input is well-formed, the SETL program above computes the output value as follows. First, it selects all of the input characters that are digits and creates a tuple (ordered sequence) of the corresponding numbers in the same order. (The SETL operator `abs` is the same as the Pascal function `ord`.) It then creates a tuple of the digits multiplied by the appropriate powers of 8. (The SETL operator `#` determines the length of a tuple.) Finally, it computes the sum (using the operator `+/`) of the elements of this tuple. A key feature of this computation is that it contains two uses of the SETL tuple former construct:

`[`*element*`:` *var* `in` *tuple* `{|` *selector*`}]` .

To illustrate how this construct is used in the program above, consider the following examples.

```
[abs C - abs '0': C in "  124 " | C/=' '] returns [1,2,4]
[D*8**(#Digits-i): D in [1,2,4](i)] returns [64,16,4]
+/[64,16,4] returns 84
```

Partly because it has a somewhat cryptic and unfamiliar syntax, a significant amount of training is required in order to use SETL's features to full advantage. However, it is possible to write SETL programs that are very compact and readable. In addition, unlike the other specifications discussed above, a SETL program is complete since it is directly executable. For example, it specifies that overflow will occur if the number represented by the input is too large.

The Refine language (Abraido-Fandino, 1987) is in many ways similar to SETL. The GIST language (Feather and London, 1982) is representative of a more ambitious direction in research on very high level languages. The goal of GIST is to provide the expressiveness of natural language while imposing formal syntax and semantics. Two examples of capabilities provided in GIST that distinguish it from SETL-like languages are: historical reference (the ability to refer to past process states) and constraints (restrictions on acceptable system behavior in the form of global declarations). For a general discussion of other desirable features of a specification language, see Balzer (1985).

Not surprisingly, the more advanced features that you put into a very high level language, the harder it is to compile (and to some extent, understand; see Swartout (1983)). Compilers for SETL and Refine have both been implemented and commercially distributed. In contrast, although it has been an active area of research, a complete GIST compiler has not yet been constructed.

When compiling languages such as SETL and Refine, the paramount problem is deciding how to implement the abstract data structures in the program. These decisions are needed not only to represent the data, but also to decide how to use loops to implement quantifiers in the program. Some researchers (e.g., Low (1978) and Rowe and Tonge (1978)) have focused on the problem of data structure implementation separate from the details of any specific very high level language.

The SETL compiler operates in a procedural fashion somewhat similar to a conventional optimizing compiler. It is only moderately successful at producing efficient code. In order to make implementation choices that lead to more efficient run-time performance, the SETL compiler will have to perform a deeper analysis of the input program. Meanwhile, a declaration language is provided that the programmer can use to tell the SETL compiler how to implement particular sets. Although this is an appealing compromise in the spirit of incremental automation, it is less than satisfactory in practice. The problem is that if the full expressive power of SETL is used in writing a program, it can be very difficult for a programmer to figure out what to recommend.

The currently favored technique for compiling very high level languages is to use a program transformation system to remove the very high level constructs. For example, this approach is used for the Refine language. As discussed in Section 3.3, transformational systems require advice on what transformations should be applied where. Unfortunately, as in the case of explicit declarations, it can be very difficult for a programmer to come up with the needed advice.

Another important trend in very high level languages is toward specialized languages for particular application areas. For applications like business data processing (Cheng et al., 1982), quite high level languages have been developed that can be successfully compiled using reasonably straightforward techniques.

2.6 Other Communication Issues

The following are three general issues that apply to any communication medium for automatic programming.

2.6.1 Wide spectrum

What should users do when they want to say something more detailed than the abstraction level supported by the input medium? The purists' answer is that it is a bad idea for users to say such things—the automatic programming system should be left to make all such decisions. However, pragmatists realize that (at least for the foreseeable future) automatic programming systems cannot operate without getting a certain amount of advice at all levels.

A good way to support this pragmatic approach is to provide users with a *wide-spectrum* language, i.e., one that provides both high level and low level constructs in a single coherent framework. For example, a very high level language like SETL is wide spectrum because it retains the features of current high level languages—e.g., it retains the ordinary looping mechanisms while adding quantification. English is inherently wide spectrum unless the vocabulary is severely restricted. In contrast, special-purpose languages are not typically wide spectrum. (The ideal automatic programming system would probably be one that supports a general-purpose wide-spectrum language plus a number of narrower special-purpose languages.)

2.6.2 Large vocabulary

In principle, any of the languages associated with the bottom-up approach to automatic programming can make use of a predefined vocabulary of terms. For example, a logical specification can refer to any number of predefined predicates and functions.

Similarly, a program written in a very high level language can refer to any number of predefined subroutines.

However, as discussed in Section 4, each of these languages implicitly limits what can be defined as a vocabulary item. In addition, research on these languages has generally focused on providing a relatively small set of very powerful primitives. An alternate approach is to make the utilization of an extensive, predefined (including domain specific) vocabulary be the primary goal. This can be thought of as an attempt to preserve the vocabulary intensive nature of English while still getting rid of most of the informality.

As an example of what a vocabulary intensive language might look like, consider the following, which is similar to descriptions produced and used in the Programmer's Apprentice.

```
EvalOctal is a function from string S to integer V.
EvalOctal implements a regular expression recognizer where:
  The recognized expression is "blank* {0|1|2|3|4|5|6|7}+ blank*".
  The failure output is -1.
  The success output, V, is the decimal value of the octal number
    in the non-blank substring of S.
```

The underlined words and phrases above are all taken to be predefined terms. This specification of `EvalOctal` is reminiscent of the English description above. However, it is different in several important ways. To start with, it uses a highly simplified syntax, including indentation to show hierarchical relationships between the parts of the specification. Also, this specification is not informal: Everything is spelled out in detail including exactly what it means for the input to contain an octal number. The feature which the specification above does share with English is the use of a large vocabulary.

Given a choice, most users would be much happier to use an awkward medium in which almost everything they want is already defined, rather than an otherwise convenient medium in which everything needs to be defined from first principles.

2.6.3 Dialogue

Because of the inherently iterative nature of the programming process, a medium must be able to support a *dialogue* between the user and the automatic programming system. In particular, the medium of communication must be capable of expressing "meta level" information, e.g., information about changes to the state of knowledge. One can imagine how natural language would serve well as a dialogue medium. Restricted notations, such as very high level languages, are clearly not sufficient by themselves.

As an illustration of the importance of supporting dialogue between the user and an automatic programming system, suppose that an automatic programming system has been given the vocabulary intensive specification of `EvalOctal` in the preceding section. While attempting to write the corresponding program, it would be helpful if the system remarked about the problem with overflow and gave the user a chance to fix it, as shown below.

```
System: There does not appear to be any limit on how large the output
        can be.  However, the largest integer value is 32767.
  User: Have EvalOctal return -1 if overflow occurs.
```

Interactive dialogue brings two new features into the automatic programming picture. First, interaction makes it possible to fix mistakes. If users are going to be brief, they will inevitably run the risk of being misunderstood. An interactive mode of operation gives users immediate feedback on the effects of their statements and allows them to make clarifications. (From this point of view, batch systems are just interactive systems with frustratingly long communication delays. Almost any system can benefit from becoming more interactive.)

Second, interaction makes it natural for users to talk about the *process* of programming as well as about what the target program should do. For example, a user might ask the system to change the target program in a certain way or to forget about some particular thing that was said earlier. This kind of communication is a prominent feature of the way programmers talk to each other, but is not captured by specification languages that merely address what target program is supposed to do.

For example, suppose that at a later time the user wanted to change the `EvalOctal` program. It would be desirable if the change could be made by talking about the program at a high level rather than using, say, a text editor. In addition, the system should respond by indicating problems caused by the change.

```
  User: Change EvalOctal so that an optional minus sign is allowed to
        appear before the first digit of the input.
System: The failure value -1 is no longer guaranteed to be distinct
        from the set of outputs corresponding to valid inputs.
```

3 How Does the System Work?

Automatic programming systems map a configuration of domain-specific terms (a requirement stated in terms of one of the input mediums above) into a configuration of implementation-specific terms (a program). Four mechanisms currently being pursued as the basis for such systems are procedural methods, deductive methods, transformational methods, and inspection methods.

3.1 Procedural Methods

To date, the most successful approach has been to simply write a special-purpose program that gets the right results. For example, other than the parsing components, most current compilers and program generators are essentially procedural in nature (although some use transformations to some extent).

The big advantage of procedural methods is that they let you get off the ground fast. It is very seldom difficult to support the first few desired features. Furthermore, you can always (try to) modify the code to support any particular additional feature.

Unfortunately, as more and more features are added to a procedural system, you reach a point of rapidly diminishing returns, because the system becomes progressively more difficult to modify. As a result, it is unlikely that the procedural approach can support the broad-coverage, end-user-oriented automatic programming systems of the future.

3.2 Deductive Methods

The problem of synthesizing a program satisfying a given specification is formally equivalent to finding a constructive proof of the specification's satisfiability. This fundamental idea underlies the deductive approach to automatic programming (Manna and Waldinger, 1980a). In principle, any method of automated deduction—resolution, natural deduction, reasoning about anonymous individuals—can be used to support automatic programming. Unfortunately, in practice, none of these methods are yet capable of proving the kinds of complex theorems required to synthesize programs of realistic size.

Deduction is basically a problem of searching for an inference path from some initial set of facts to a goal fact. The search is exponential in nature because at every step there are many ways for inference rules to be applied to facts. Current deductive systems cannot discover complex proofs because they are unable to effectively control the search process.

To deal with this control problem, deductive systems typically must adopt the assistant approach—that is, they seek advice from the user. Unfortunately, users who want to avoid programming probably want to avoid theorem proving as well!

An even more fundamental problem with the deductive approach is that it is at odds with the need for an automatic programming system to make a good faith effort to satisfy the "spirit" of a requirement. For example, the theorem-proving process contains no bias toward finding the proof corresponding to the most efficient program, or even a reasonably efficient program.

Despite these limitations, deductive methods have several advantages. In particular, they are very general and quite effective, as long as they are limited to proving simple theorems. As a result, deductive methods are certain to play an important role in the automatic programming systems of the future. The challenge is to combine automated deduction with other methods so that its inherent limitations can be avoided.

3.3 Transformational methods

Transformational implementation systems (e.g., TI (Balzer, 1985) and PDS (Cheatham, 1984)) have dominated research in automatic programming. In this approach, the input to the automatic programming system is a program written in a very high level language. A sequence of transformations is applied to convert this input into a low-level implementation.

A transformation has three parts: a pattern, a set of logical applicability conditions, and an action procedure. When an instance of the pattern is found, the logical applicability conditions are checked to see whether the transformation can be applied. If the applicability conditions are satisfied, the action can be evaluated to compute a new section of code, which is used to replace the code matched by the pattern. Typically, transformations are *correctness preserving*, meaning that the matched code and its replacement represent logically equivalent computations.

There are two basic kinds of transformations. Some transformations (often called *vertical* transformations) replace specification-like constructs (e.g., quantification over a set) with conventional constructs (e.g., iterating over a list). These transformations encode knowledge of how to implement algorithms and data structures. Other transformations (often called *horizontal* transformations) perform rearrangements and optimizations (e.g., moving an unchanging computation out of a loop), which do not change the level of abstraction. In practice, these two kinds of transformations are interleaved in long sequences, passing through multiple levels of abstraction.

The central feature of transformational methods is the *transformational rewrite cycle*. The state of the transformation process is represented as a program in a wide-spectrum representation that is capable of expressing both the user's input and the final program. On each cycle, a transformational system selects a transformation and applies it to some place in the program. The cycle continues, accumulating the results of longer and longer chains of transformations, until some termination condition is satisfied (e.g., until there are no more very high level constructs).

As an example of how transformational implementation works, let us follow the steps by which part of the SETL program in Section 2.5 could be transformed into an efficient low level program. In particular, we focus on the statement in `EvalOctal` that computes the value of a tuple of integers viewed as an octal number:

```
return +/[D*8**(#Digits-i): D in Digits(i)];
```

The two most important transformations that need to be applied here are vertical transformations which define how the summation and tuple formation operations can be implemented. Each of these transformations is defined below in a simplified, informal notation in which the applicability conditions are expressed in English and the action procedure is shown in the form of a template for the resulting code. (Italics in the pattern and the result fields denote variables that match against arbitrary parts of the program. `<Temp>` stands for a temporary variable name generated by the action procedure.)

```
      name: tuple-former
   pattern: X := [E: V in T(I) | P];
conditions: X, V, T, and I are distinct variables.
            X does not appear in E, T, or P.
    result: X := [];
            for I := 1 to #T;
              if P then X := X with E;
            end for;

      name: summation
   pattern: X := +/T;
conditions: X and T are distinct variables.
    result: X := 0;
            for <Temp> := 1 to #T;
              X := X+T(<Temp>);
            end for;
```

Although these two transformations capture the essence of the implementation of the `return` statement above, they cannot be directly applied to the statement because their patterns do not exactly match: Both transformations expect to match against assignment statements; the `tuple-former` transformation expects there to be a selection predicate, which is missing in the statement above. (These kinds of problems are typical.)

In order to apply the two vertical transformations above, the following two horizontal transformations first need to be used. The `factor-out` transformation pulls a subexpression out of a statement and creates a separate assignment statement. The `add-tuple-selector` transformation introduces a degenerate selection predicate (`true`) into a tuple former which does not contain a selection predicate.

```
      name: factor-out
   pattern: S;
conditions: S is a statement containing a subexpression E.
            E can be evaluated separately before the rest of S.
    result: <Temp> := E;
            S <with Temp substituted for E>;

      name: add-tuple-selector
   pattern: [E: V in Expr]
conditions: none
    result: [E: V in Expr | true]
```

Applying `factor-out` twice and `add-tuple-selector` once produces:

```
Temp1 := [D*8**(#Digits-i): D in Digits(i) | true];
Temp2 := +/Temp1;
return Temp2;
```

The vertical transformations can then be applied producing:

```
Temp1 := [];
for i := 1 to #Digits;
  D := Digits(i);
  if true then Temp1 := Temp1 with D*8**(#Digits-i);
end for;
Temp2 := 0;
for Temp3 := 1 to #Temp1;
  Temp2 := Temp2+Temp1(Temp3);
end for;
return Temp2;
```

Although correct, the code produced above is far from satisfactory. The biggest problem is that it slavishly follows the details of the initial SETL code. In order to achieve reasonable efficiency, the two loops have to be merged, eliminating the computation of the intermediate tuple `Temp1`. This can be done using the transformation below.

```
      name: serial-for-loop-combination
   pattern: V := [];
            for I := 1 to L;
              S1
              V := V with E;
              S2
            end for;
            for J := 1 to #V;
                S3
            end for;
conditions: V, I, J, and E are distinct variables.
            S1, S2, and S3 are zero or more statements.
            S1, S2, and S3 cannot cause loop termination.
            S1 and S2 cannot change I.
            S3 cannot change J.
            S1 and S2 do not refer to V.
            Every reference to V in S3 is of the form V(J).
            There are no references to V in the surrounding program.
    result: for I := 1 to L;
              S1
              <Temp> := E;
              S2
              S3 <with Temp substituted for V(J)>
            end for;
```

As was the case above, this transformation is not quite applicable to the program as it stands. First, the following horizontal transformations have to be applied in order to get rid of the unnecessary `if` statement and to move the assignment statement out from between the loops.

```
      name: remove-if
   pattern: if true then S;
conditions: none
    result: S;
```

```
      name: interchange
   pattern: S1; S2;
conditions: S1 and S2 are statements which do not interact.
    result: S2; S1;
```

In addition, the `factor-out` transformation must be used to introduce an additional variable. This yields:

```
Temp2 := 0;
Temp1 := [];
for i := 1 to #Digits;
  D := Digits(i);
  Temp4 := D*8**(#Digits-i);
  Temp1 := Temp1 with Temp4;
end for;
for Temp3 := 1 to #Temp1;
  Temp2 := Temp2+Temp1(Temp3);
end for;
return Temp2;
```

The loops can then be combined yielding:

```
Temp2 := 0;
for i := 1 to #Digits;
  D := Digits(i);
  Temp4 := D*8**(#Digits-i);
  Temp2 := Temp2+Temp4;
end for;
return Temp2;
```

In order to get really efficient code, the transformation process must go one step further by simplifying the entire computation using Horner's rule.

```
      name: Horner's-rule
   pattern: V := 0;
            for I := 1 to #T;
              S1
              V := V+T(I)*X**(#T-I);
              S2
            end for;
conditions: V, I, and T are distinct variables.
            S1 and S2 are zero or more statements.
            S1 and S2 cannot cause loop termination.
            S1 and S2 cannot change I.
            S1 and S2 do not refer to V.
            X is an expression whose value is invariant in the loop.
    result: V := 0;
            for I := 1 to #T;
              S1
              V := V* X+T(i);
              S1
            end for;
```

As usual, additional horizontal transformations must be applied before this transformation can be used. In particular, a `factor-in` transformation (which is the inverse of `factor-out`) must be applied to get the computation into the right form. Once this is done, the following code can be produced.

```
Temp2 := 0;
for i := 1 to #Digits;
  Temp2 := Temp2*8+Digits(i);
end for;
return Temp2;
```

At this point good code has been produced for just one statement in the SETL program `EvalOctal`. Many more transformations would be needed to translate the other statements and then to combine the results of all these statements together into an efficient program.

It is also important to realize that, like most textbook illustrations of program transformation, this example is misleading because it ignores the control problem. At each step above there would be a host of other transformations which could be applicable, but would not be productive to apply. For example, many horizontal transformations, such as `factor-out` and `factor-in`, can be applied in almost any situation. It is easy for a transformation system to get lost in aimless sequences of such transformations.

It is also easy to misapply vertical transformations. For example, before applying `Horner's-rule` it would have been easy to make the mistake of moving the invariant computation `#Digits` out of the loop. Equally well, you might apply a strength reduction transformation to simplify the computation of `D*8**(#Digits-i)` in isolation. Either transformation would have blocked the later application of `Horner's-rule`.

In many ways, sequences of transformation steps are not that different from sequences of proof steps. Therefore, it is not surprising that transformational implementation systems suffer from essentially the same control problem as automatic theorem provers. As a consequence, transformational systems must either seek advice from the user or place strong restrictions on the kinds of transformations that can be used. Unfortunately, advice-taking transformational systems are not much more satisfactory than advice-taking deductive systems and have not yet made it out of the laboratory. However, restricted transformational modules can be found as components of a variety of compilers and other systems.

A major strength of transformational methods is that they provide a very clear representation for certain kinds of programming knowledge, such as Horner's rule. For this reason, transformational methods in some form are certain to be part of all future automatic programming systems.

3.4 Inspection Methods

Human programmers seldom think only in terms of primitive elements, such as assignments and tests. Rather, like engineers in other disciplines, they think mostly in terms of clichéd combinations of elements corresponding to familiar concepts. *Successive approximation*, *interrupt-driven architecture*, and *information system*, are examples of clichés spanning the range from low-level implementation ideas to high-level specification concepts.

Given a knowledge of clichés, it is possible to perform many programming tasks *by inspection* rather than by reasoning from first principles. For example, in analysis by inspection, properties of a program are deduced by recognizing occurrences of clichés and referring to their known properties. In synthesis by inspection, implementation decisions are made by recognizing clichés in specifications and then choosing among various clichéd implementations. By using global understanding, inspection methods reduce the search control problems that arise with other methods.

The central feature of inspection methods is the codification and use of *clichés*. A cliché has three parts: a skeleton that is present in every occurrence of the cliché, roles whose contents vary from one occurrence to the next, and constraints on what can fill the roles. An essential property of clichés is their interrelationships. For example, a cliché may specialize or extend another cliché. Algorithmic and data structure clichés implement specification clichés. These relationships are the driving force behind analysis and synthesis by inspection.

As with deductive and transformational methods, it has not yet been shown that inspection methods can be automated without advice from the user. However, when used with the assistant approach to automatic programming, inspection methods have an important advantage: A shared vocabulary of clichés is a natural medium in which to communicate explanations and advice between the system and the user.

The Programmer's Apprentice project has demonstrated the automation of several aspects of inspection methods, including: a system that automatically analyzes a program to identify the algorithmic clichés used (Wills, 1990; Wills, 1992); an intelligent assistant for defining software requirements using clichés (Section 5.1); and a knowledge-based editor for constructing programs using clichés (Section 5.3).

In human programming, inspection methods are the most effective approach to use whenever they are applicable. However, since inspection methods are ultimately based on experience, they are applicable only to the routine parts of programming problems. As a result, inspection methods must be used as part of a hybrid strategy that falls back on more general methods such as deduction and transformation when inspection fails.

4 What Does the System Know?

No matter what mechanism is used inside an automatic programming system, the system must have at least an implicit knowledge of domain clichés (so that it can interpret the terms used by the user) and of programming clichés (so that it can produce programs without endlessly "reinventing the wheel"). Whether knowledge of clichés is represented procedurally, logically, transformationally, or in some other way, the benefits of automatic programming can be traced almost exclusively to the productivity and reliability benefits of reusing this knowledge. The following examples of programming clichés illustrate the diversity of knowledge required:

a. *Matrix add*—the algorithm for adding together two matrices. This cliché is independent of the data representation of the matrices and the type of number stored in the matrices.

b. *Stack*—the data abstraction and its associated operations. Both the representation and the operations are independent of the type of stack element.

c. *Filter positive*—selecting the positive elements of a temporal sequence of quantities in a loop. For example, in the code fragment below, the `if` statement implements a filter positive.

```
do ...
  X = ...;
  if X>0 then ... X ...;
end;
```

   This cliché is independent of the type of number in the sequence and how the sequence is generated.

d. *Master file system*—a cluster of programs (reports, updates, audits, etc.) that operate on a single master file, which is the sole repository for information on some topic. This cliché is essentially a set of constraints on the programs and how they interact with the file. It is independent of the kind of data stored in the file and the details of the computation performed by the programs.

e. *Deadlock free*—the property of a set of asynchronously interacting programs that guarantees they will not reach a state where each program is blocked waiting for some other program to act. This cliché places restrictions on the ways in which the programs can interact. However, it is independent of the details of the computations performed by the programs.

f. *Move invariant*—moving the computation of an expression from inside a scope of repetitive execution to outside the repetitive scope, so long as it has no side effects and all of the values it references are constant within the repetitive scope. This idea is independent of the specific computation being performed by the expression and by the rest of the repetitive scope. In addition, the idea is applicable to both loops and recursive programs.

The clichés above differ along many dimensions. Matrix add is primarily computational, while stack is data-oriented. Matrix add can be used in a program as a module, while filter positive is fragmentary and must be combined with other fragments to be useful. Matrix add, stack, and filter positive are all relatively low-level, localized clichés. In contrast, master file system and deadlock free are high level and diffuse.

Representing and using such a wide variety of clichés in an automatic programming system is a major challenge. The following are the main desiderata for a suitable knowledge representation.

a. *Expressiveness.* The representation must be capable of expressing as many different kinds of clichés as possible. An important though hard to assess aspect of this is desideratum is whether a formalism is merely *capable* of representing a given cliché (e.g., via some obscure circumlocution) or whether it represents it in a straightforward way which supports the other desiderata as well.

b. *Convenient combination.* The methods of combining clichés must be easy to implement and the properties of combinations should be evident from the properties of the parts. It is also desirable for the representation to provide a fine "granularity," so that each cliché embodies only a single idea or design decision and users have the maximum possible freedom to combine them as they choose.

c. *Semantic soundness.* The representation must be based on a mathematical foundation that allows correctness conditions to be stated. The semantic basis does not necessarily need to support totally automatic verification. Although less convenient, machine-aided (or even manual) verification may be sufficient in many situations.

d. *Machine manipulability.* It must be possible to manipulate the representation effectively using computer tools. In order to effectively deal with a large library of clichés, tools need to be developed to support the automatic creation, modification, selection, and combination of clichés.

e. *Programming language independence.* The representation should not be tied to the syntax of any particular programming language. The most obvious benefit of a language independent formalism is that clichés can be reused in many different language environments. However, an equally important reason for requiring language independence is that it facilitates abstracting away from the details of how, for example, data and control flow are implemented with specific syntactic constructs.

The following sections discuss a number of approaches to the representation of programming clichés. The relative strengths and weaknesses of the approaches are evaluated in the light of the five desiderata above. The central theme which ties these sections together is the search for representations that are capable of expressing the wide range of components desired without sacrificing the other desiderata. Figure 4 summarizes graphically the major flow of ideas between the approaches discussed.

As a point of comparison for other formalisms, one must consider free-form English text. Much of the knowledge which needs to be formalized is already captured informally in the vocabulary of programmers and in textbooks on programming. The great strength of English text is expressiveness. It is capable of representing any kind

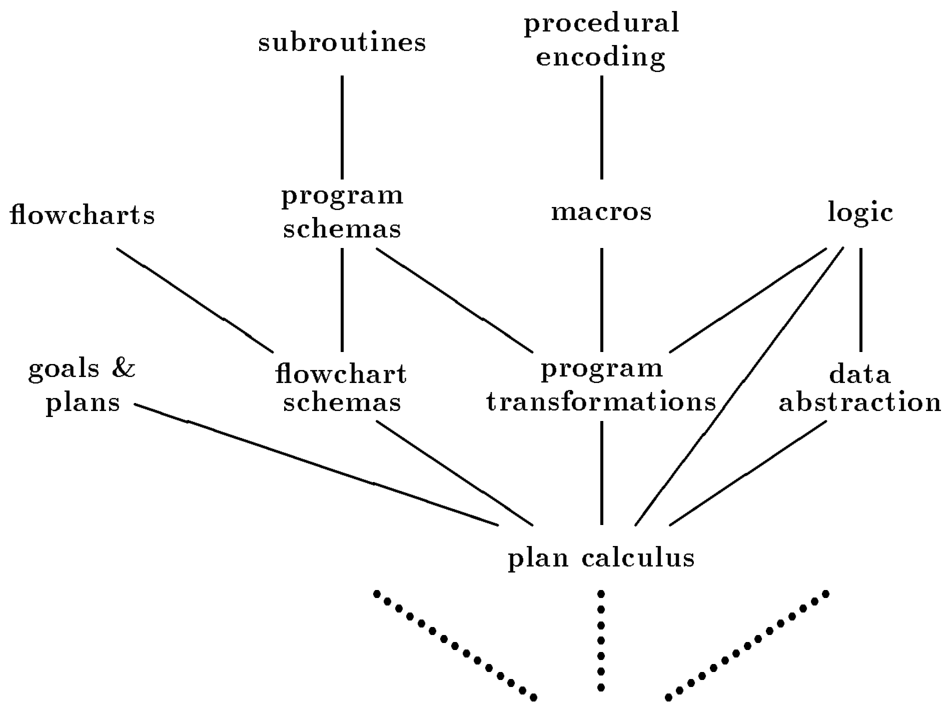

Figure 4. Inheritance of ideas among the major approaches that have been used to represent programming knowledge.

of cliché. Moreover, it is programming language independent. Unfortunately, English text does not satisfy any of the other desiderata. There is no theory of how to combine textual fragments together; there is no semantic basis that makes it possible to determine whether or not a piece of English text means what you think it means; and free-form English text is not machine manipulable in any significant way.

4.1 Subroutines

Subroutines have many advantages as a representation. They can be easily combined by writing programs that call them. They are machine manipulable in that high-level language compilers and linkage editors directly support their combination. Further, they have a firm semantic basis via the semantics of the programming language they are written in.

Unfortunately, subroutines are limited in their expressiveness. They are really only convenient for expressing localized computational algorithms such as matrix add. They cannot represent data clichés such as stack, fragmentary clichés such as filter positive, diffuse high-level clichés such as master file system, or transformational clichés such as move invariant. In addition, they are not fine-grained. It is difficult to write a subroutine without gratuitously specifying numerous details that are not properly part of the cliché. For example, in most languages, there is no convenient way to write a subroutine representing matrix add without specifying the data representation for the matrices and the numbers in them.

4.2 Macros

A subroutine specifies a fixed piece of program text. The only variability allowed is in the arguments that are passed to the subroutine. In contrast, a macro can include an arbitrary computation that creates a piece of program text. Due to the provision for arbitrary computation, macros are a considerable improvement over subroutines in expressiveness. They can be used to represent data clichés and fragmentary clichés. In addition, they can represent clichés at a much finer granularity. For example, it is straightforward to write a macro which represents matrix add independent of the data structures it operates on. Note however, that macros are still not suited to representing diffuse clichés or transformational ones.

Like subroutines, macros are machine manipulable in that macro processors directly support the evaluation of macro calls and the integration of the resulting program text into the program as a whole. Unfortunately, macros are less satisfactory than subroutines in other respects. Though macro calls are combined syntactically in essentially the same way as subroutine calls, their combination properties are not as simple. For example, since a macro can perform arbitrary computation utilizing its calling form in order to create the resulting program text, there is no guarantee that nested macro calls will operate as they are intended. The macro writer must take extreme care in order to insure that flexible combination is possible. This unfortunately militates against the increased expressiveness which is the primary advantage of macros.

The most significant problem with macros is that they lack any firm semantic basis. Because they allow arbitrary computation, it is very difficult to verify that a macro accurately represents a given cliché. It is even more difficult to show that a pair of macros can be combined without destructive interaction.

4.3 Program Schemas

There has been a considerable amount of theoretical investigation into the use of program schemas to represent programming knowledge (Basu and Misra, 1976; Gerhart, 1975; Wirth, 1973). Program schemas are essentially templates with holes in them which can be filled with program text by the user. As such, they can be viewed as a compromise between subroutines and macros. The main improvement of program schemas over macros is that, like subroutines, they have a firm semantic foundation in the semantics of the programming language they are written in and their combination properties are relatively straightforward.

Unfortunately, though program schemas are an improvement in expressiveness over subroutines, they are significantly less expressive than macros. Like macros, program schemas are of some use in representing data clichés such as stack and can represent clichés at a finer granularity than subroutines. In addition, they could be used to represent matrix add independent of the representation for the matrices involved. However, unlike macros, program schemas cannot in general be used to represent fragmentary clichés such as filter positive. They are no more useful than macros at representing diffuse or transformational clichés.

4.4 Flowcharts and Flowchart Schemas

A limitation shared by subroutines, macros, and program schemas is programming language dependence. One way to alleviate this problem would be to write clichés in a programming language independent representation, such as a flowchart. Flowcharts use boxes and control flow arrows in order to specify control flow independent of any particular control flow construct. Similarly, data flow arrows can be used to represent data flow independent of any particular data flow construct.

A flowchart using data and control flow arrows is basically equivalent to a subroutine and has the same level of expressiveness. In analogy to program schemas, one can gain additional expressiveness by using flowchart schemas (Ianov, 1960; Manna, 1974)—flowchart templates with holes in them where other flowcharts can be inserted. A flowchart language can be given a semantic foundation similar to that of a programming language. In addition, flowcharts and flowchart schemas can be combined in the same semantically clean way that subroutines and program schemas can be.

Flowcharts and flowchart schemas are a significant improvement over subroutines and program schemas in that they are programming language independent. However, with regard to the other desiderata, they are basically identical to subroutines and program schemas. In particular, they are no more expressive.

4.5 Logical Formalisms

With the exception of some macros, the representations discussed above are all *algorithmic* in that they represent a cliché by giving an example (or template) of it in a programming (or flowchart) language. In addition, the only way to use a cliché is to place it somewhere in a program. This fundamentally limits the expressiveness of these representations.

The extensive work on specifying the semantics of programming languages suggests a completely different approach to the problem of representing clichés, i.e., using logical formalisms such as predicate calculus. A key advantage of logical formalisms is semantic soundness.

Another important advantage of logical formalisms is in the area of expressiveness. In contrast to the algorithmic representations, logical formalisms have no trouble representing diffuse, high-level clichés such as master file system and deadlock free. The usefulness of such clichés is enhanced by the fact that logical formalisms also have very convenient combination properties. Specifically, the theory generated by the union of two axiom systems is always either the union of the theories of the two cliché systems or a contradiction, but never some third, unanticipated theory. An additional advantage of logical formalisms is that they are inherently programming language independent.

However, logical formalisms are quite cumbersome when it comes to representing an algorithmic cliché such as matrix add. Given a cliché such as stack, which combines some non-algorithmic aspects with some algorithmic aspects, logical formalisms are convenient for the former, but not the latter. This suggests that logical formalisms are best used as an adjunct to, rather than a replacement for, algorithmic representations.

The greatest weakness of logical formalisms is in the area of machine manipulability. It is not hard to represent logical formulas in a machine manipulable way. However, at the current state of the art, practical automatic theorem provers are only capable of relatively simple logical deductions. As a result, it is hard to do anything useful with logical formulas. For example, if a programming system were to be based on the combination of clichés represented as logical formulas, the system would need to have a module which could produce program text corresponding to sets of logical formulas. Unfortunately, although this kind of automatic programming has been demonstrated on small examples (Manna and Waldinger, 1980a), it has not yet progressed to the point where it is at all practical.

4.6 Data Abstraction

An interesting area of inquiry which has combined logical and algorithmic representations is data abstraction. The contribution of data abstractions is that they extend the expressiveness of algorithmic formalizations into the realm of clichés with data structure aspects. For example, data abstraction can be used to represent stack in full generality and to represent matrix add without specifying the data representation to be used for the matrices.

A considerable amount of research has been done on how to state the specifications for a data structure and its associated access functions, which provides a semantic basis for data abstractions and for methods of combining them. In addition, languages such as Alphard (Wulf et al., 1976), CLU (Liskov and Guttag, 1986), and Ada (1983) have been developed with constructs that directly support data abstraction. This demonstrates the ease with which data abstractions can be represented in a machine manipulable (though language dependent) form.

4.7 Program Transformations

As mentioned in Section 3.3, a program transformation can be thought of as a representation of a piece of programing knowledge. From this point of view, the most interesting contribution of transformations is that they view program construction as a *process*. Rather than viewing a program solely as a static artifact which may be decomposed into clichés the way a house is made up of a floor, roof, and walls, transformations view a program as evolving through a series of construction steps which utilize clichés which may not be visible in the final program, just as the construction of a house requires the use of scaffolding and other temporary structures. This point of view enables transformations to express clichés such as move invariant, which are common steps in the construction of a program rather than common steps in the execution of a program.

Another important aspect of transformations is that they can be combined in a way which is quite different from the other representations. As mentioned above, many simple transformations are basically just macros which specify how to implement particular high-level constructs in a wide spectrum language. These transformations are only triggered when instances of their associated high-level constructs appear; thus they only operate where they are explicitly requested and combine in much the same way as macros.

However, other transformations are much less localized in the way they operate. For example, a transformation representing move invariant would have applicability conditions (e.g., that the expression is invariant) that must look at large parts of the program. In addition, such transformations are not intended to be applied only when explicitly requested by the user. Rather, they are intended to be used whenever they become applicable for any reason. This makes powerful synergistic interaction between transformations possible.

Unfortunately, if transformations are allowed to contain arbitrary computation in their actions, they have the same difficulty with regard to semantic soundness and convenient combinability that macros have. The transformation writer has to take great care in order to insure that the interaction between transformations will in fact be synergistic rather than antagonistic. In order to have a semantic basis, transformations must include a logical description of what the transformation is doing. One important way that this has been done is to focus on transformations which are correctness preserving—ones which, from a logical perspective, do nothing.

A difficulty with transformations is that, as generally supported, they are very much programming language dependent. This not only limits the portability of clichés represented as transformations, it also limits the way transformations can be stated by requiring that every intermediate state of a program being transformed has to fit into the syntax of the programming language. One way to alleviate these problems would be to apply transformations to a programming language independent representation such as flowcharts.

The Plan Calculus (described in more detail Section 5.2) combines ideas from many of the representations described above. It achieves programming-language independence through the use of data flow and control flow notions from flowchart schemas. It uses aspects of logic and data abstraction to represent data invariants and other diffuse aspects of clichés. It uses goals and plans to keep a record of the design decisions in a program. It includes the concept of language-independent, bidirectional program transformations linking pairs of flowchart schemas.

5 The Programmer's Apprentice

The long-term goal of the Programmer's Apprentice project (Rich and Waters, 1988; Rich and Waters, 1990) was to develop a theory of how expert programmers analyze, synthesize, modify, explain, specify, verify, and document programs. This is basic research at the intersection of artificial intelligence and software engineering. From the perspective of artificial intelligence, we used programming as a domain in which to study fundamental issues in knowledge representation and reasoning. From the perspective of software engineering, we applied techniques from artificial intelligence to automate the programming process.

Recognizing that it would be a long time before it was possible to fully duplicate human programming abilities, the near-term goal of the project was to develop a system, called the Programmer's Apprentice, that provides intelligent assistance in all phases of the programming task.

This section describes two parts of the Programmer's Apprentice that have been implemented (as demonstrations) and some of the underlying knowledge representation techniques used.

5.1 The Requirements Apprentice

The Requirements Apprentice (RA) is an intelligent assistant for the acquisition and analysis of software requirements. The focus of the RA is on the *formalization* phase of software requirements, i.e., the process by which informal descriptions become formal ones. The kinds of informality the RA deals with include: abbreviation, ambiguity, poor ordering, contradiction, incompleteness, and inaccuracy.

Figure 5 shows the role of the RA in relation to other agents involved in the requirements process. Note that the RA does not interact directly with an end user, but is an assistant to a requirements analyst.

The RA produces three kinds of output. Interactive output notifies the analyst of conclusions drawn and problems detected as information is being entered. A requirements knowledge base represents everything the RA knows about the evolving requirement. Finally, the RA can create a more or less traditional requirements document summarizing the current state of the knowledge base.

Internally, the RA is composed of three parts: A system called Cake (Rich and Feldman, 1992) provides the basic knowledge representation and automated reasoning facilities. The *executive* (Exec) contains algorithms and data structures that are specific to the RA and provides control of reasoning for Cake. The *cliché library* contains reusable fragments of requirements and associated domain models, represented as a frame hierarchy. Figure 6 shows a fragment of this library relevant to the example session below.

The example session is based on a requirements benchmark (Babb, 1985) dealing with the specification of a university library data base. Although the RA's cliché library contains many concepts that will be relevant, the RA doesn't know anything

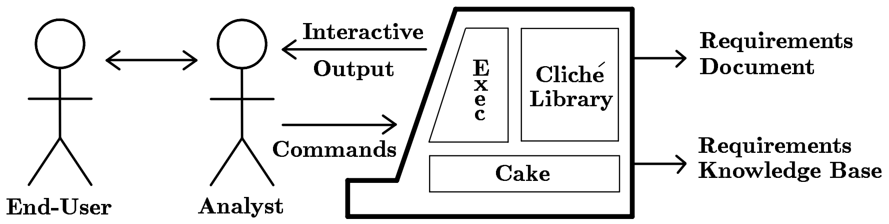

Figure 5. Architecture of the Requirements Apprentice.

about libraries per se at the start of the session.

To avoid distraction with the syntactic details of the RA's input language, what the user types (following the `>` prompt) below has been replaced by editorial paraphrases in italics. The corresponding literal transcript can be found in (Reubenstein and Waters, 1991). Note that this demonstration system has ignored important user interface issues.

1>*"University Library Data Base"* (ULDB) *is a system.*
2>*"University Library"* (UL) *is an instance of "library".*
UL *is part of the environment.*

The distinction between the system being specified and the environment (which cannot be changed) is part of the RA's basic epistemology.

The simple initial statement of the requirement is for the data base to "track" the state of the (actual) library. There are several different versions of tracking specifications in the cliché library. The type of tracking system being specified will be refined as the session progresses.

3>ULDB *tracks* UL.
```
  ULDB Is-Instance-Of Tracking-System.
```

At this point, a number of issues are pending. Note that the RA must accept information in any order presented by the analyst (e.g., uses of words before their definition) and come to the same conclusions. The third pending issue below says that the RA needs to know the possible states of the library.

4>*Show pending issues.*
```
  1- Need-Further-Disambiguation of Tracks.
  2- Need-Definition of "Library".
  3- Need-Definition (Item-States UL).
```

A central concept in the remainder of this session is *repository*, represented as a frame in the RA's cliché library with the following slots: the *type of items* stored in it, the *patrons* that utilize it, and the *staff* that manages it. Two key operations on a repository are *adding* and *removing* items.

5>*"Library" is a kind of repository, with items of type "book."*

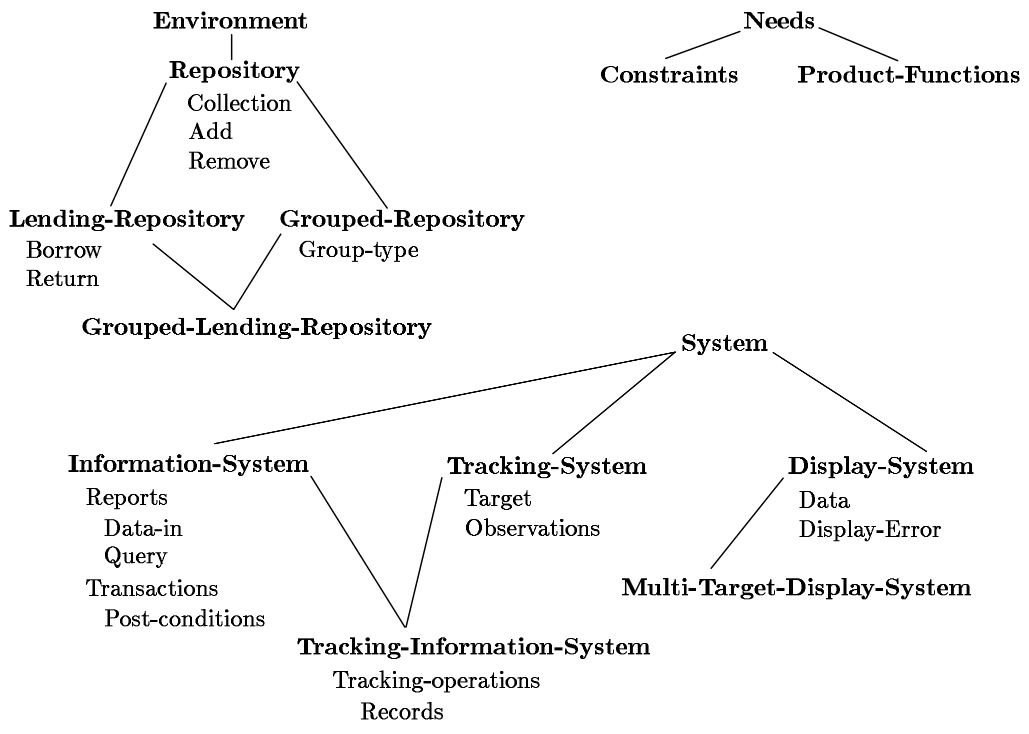

Figure 6. The structure of a fragment of the requirements cliché library.

Notice that the new type *book* is introduced above. The analyst now specifies some properties of this type.

6>*"Book" is a kind of physical object, with slots "title," "author," and "*ISBN*."*
7>*"*ISBN*" is a kind of integer.* ISBN *uniquely identifies a book.*

The analyst now begins to define various types of transactions on the data base. As more information is accumulated, the RA is able to further refine the type of ULDB to be a tracking *information* system (because it *records* information about the history of the tracked object, rather than just providing the current state).

8>*"Check out" is a transaction that records removing books from the library.*
```
ULDB Is-Instance-Of Tracking-Information-System.
```
9>*"Check in" is the inverse of check out.*
```
"Check-In" Is-Instance-Of Action-Tracking-Opn.
```

Using simple algebraic reasoning, the RA is able to deduce quite a lot of information from the statement that check in is the inverse of check out (i.e., it tracks the inverse state change).

The first serious problem detected by the RA in this session arises when the analyst defines two further transactions below.

10>*"Unshelf" is a transaction that records removing books from the library.*

11>*"Unshelf all" is a transaction that records removing books from the library. Unshelf all removes the set of all books with a given* ISBN.

```
   **Conflict #1
   **Colliding-Definitions Unshelf And Unshelf-All
```

Based on what it has been told so far, the RA is able to deduce that the *unshelf* transaction and the *unshelf all* transaction are identical: Since ISBN uniquely identifies a book, there can never be two books with the same ISBN. Therefore the set of books removed by the unshelf all transaction must contain exactly one book. The RA has a built-in expectation that terms should not be synonymous unless they are explicitly defined to be synonyms.

In response to the analyst's request for an explanation of the problem, the RA prints out a listing (mostly elided below) of the dependency tree underlying the problematic conclusion. Unfortunately, the full tree is much too large and hard to understand to be useful to the analyst in raw form.

12>*Explain.*

```
   1. (Equal-Defs Unshelf Unshelf-All) is True
      by Implication from:
      1. (Implies
          (And (Tracking-Operation Unshelf)
               (Tracking-Operation Unshelf-All)
               (= Frame-I4 Frame-I5))
          (Equal-Defs Unshelf Unshelf-All))
      2. (And (Tracking-Operation Unshelf)
              (Tracking-Operation Unshelf-All)
              (= Frame-I4 Frame-I5)) is True
         by Conjunction from:
         1. (= Frame-I4 Frame-I5) is True
            by Equality from:
  ...
   **Premises:
   1. (= (Records Unshelf-All) Remove)
   2. (= (Objects Unshelf-All)
         (!The-Set-Of-All (?B)
          Such-That (= (ISBN ?B) $Input)))
   3. (= (Records Unshelf) Remove)
   4. (Default 23
       (= (Objects Unshelf)
          (!The (?O) Such-That
                (= (ISBN ?O) $Input)))
   ...
```

A slight improvement over the full dependency tree is the list of premises shown above. Premises corresponding to statements made by the analyst are printed first, followed by assumptions that come from the cliché library. Unfortunately, even the premise list can become quite long. (In this example, the untruncated list has over twenty entries.) Generating better explanations is an important area for future research.

It is often a good idea to ignore a conflict or contradiction for a while until clarifying information comes along. If the analyst chooses to ignore this conflict, it will be added to the list of pending issues.

At this point, however, the analyst realizes that the requirement thus far reflects a token/type confusion between the notion of a physical copy of a book and the notion of a book as a class—i.e., a set of copies with the same ISBN. (This confusion actually appears in the benchmark on which this session is based.) To fix the problem, a new term denoting "copy of book" must be introduced. In addition, something has to be done about the fact that some uses of "book" in the statements above refer to the old concept and some refer to the new concept.

The RA provides the `Reformulate` command demonstrated below to assist with this kind of incremental change. The command displays all the statements that contained the term in question and asks the analyst to choose where substitution should occur. The RA then retracts the obsolete statements and asserts the new ones, relying on Cake's truth maintenance facilities to preserve all of the appropriate deductions.

```
13>Reformulate using "copy of book" in place of book.
   5."Library" is a kind of repository, with items of type "book."
     **Reformulate? >Yes

   6."Book" is a kind of physical object, with slots "title," "author," and "ISBN."
     **Reformulate? >Yes

   7."ISBN" is a kind of integer. ISBN uniquely identifies a book.
     **Reformulate? >No
```

In the interests of brevity, the session is truncated at this point.

Research on the RA addresses an important gap in much of the work on tools to support requirements analysis. Whereas the requirements acquisition process almost inevitably begins with a vague and informal statement of what is desired, most requirements analysis tools need some sort of relatively formal requirement statement as their input. The RA attacks head-on the problem of fleshing out and cleaning up a vague and informal requirement statement.

5.2 The Plan Calculus

The Plan Calculus seeks to satisfy the desiderata presented in Section 4 for representing programming clichés by combining ideas from several other approaches. The Plan Calculus contains three components: plan diagrams, logical annotation, and overlays.

A *plan diagram* (see Figure 7) contains information about the algorithmic aspects of a cliché. In order to achieve language independence, plan diagrams are represented as hierarchical data flow schemas. Computations are represented as boxes with input

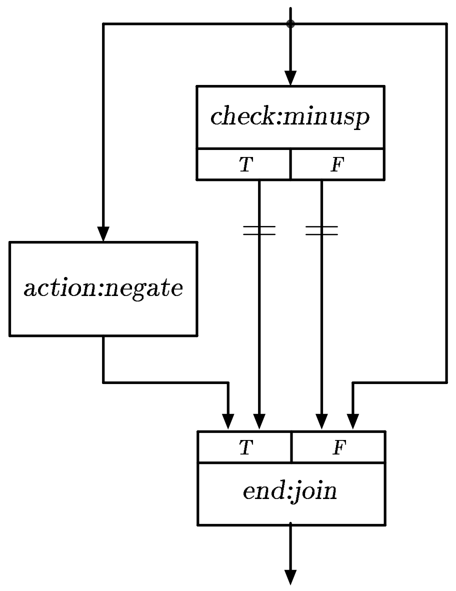

Figure 7. Plan diagram for absolute value cliché.

and output ports. Control flow and data flow are both represented using arcs between ports. Plan diagrams are hierarchical—a box in a diagram can contain an entire sub-diagram. Further, the diagrams are schematic—they can contain empty boxes (roles) which are to be filled in later.

The plan diagram in Figure 7 shows an algorithm for computing the absolute value of a number. In the figure, data flow arcs are drawn as solid lines and control flow arcs as lines with cross-hatching. The diagram is composed of an operation box (*action*) whose output is the negation of its input, a test box (*check*) which splits control based on whether or not its input is less than zero, and a "join" box (*end*) which rejoins the control split by the test. The output of the join is determined by the control flow path which is used to enter it.

The non-algorithmic aspects of a cliché are represented using predicate calculus assertions attached as annotations on a plan diagram. Each box in a plan diagram is annotated with a set of preconditions and postconditions. In addition, logical constraints between roles are used to limit the way in which the roles can be filled in. Finally, dependency links record a summary of a proof that the specifications of the plan as a whole follow from the specifications of the inner boxes and the way these boxes are connected. Clichés such as master file system and deadlock free, which have little or no algorithmic aspect, are represented by plans which consist almost entirely of predicate calculus assertions with little or no diagrammatic information.

In order to represent the combination of algorithmic and data structure clichés, the basic flowchart-like ideas behind plan diagrams have been extended to include parts which represent data objects as well as sub-computations. Data parts can be left unspecified as data roles and can be annotated with specifications, constraints, and dependencies. Given these extensions, the Plan Calculus is capable of representing

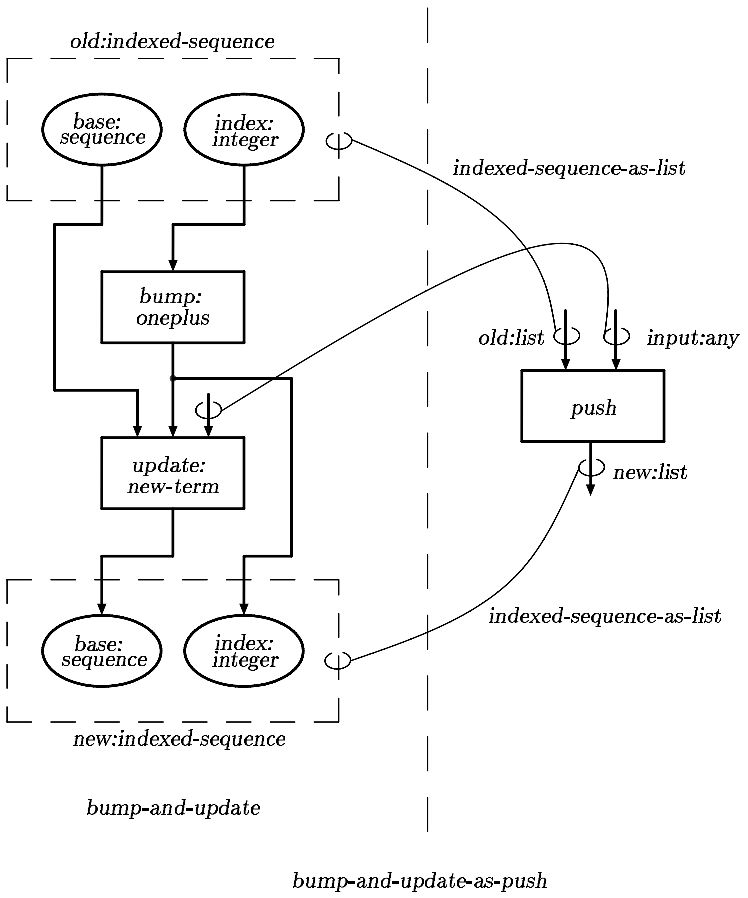

Figure 8. Overlay for implementing a push operation.

the same kinds of information as data abstraction formalisms. For example, the stack data abstraction consists of a number of logically interrelated plan diagrams, one of which represents the stack data object and the rest of which represent the operations on a stack.

The transformational aspects of programming knowledge are represented as overlays. An *overlay* (see Figure 8) is a mapping between plans. It specifies a set of correspondences between the roles of the plans. Overlays are similar to transformations in which both the left and right hand side are plans. However, overlays differ from program transformations in two ways: Overlays are bidirectional and their actions are declarative as opposed to procedural. The fact that overlays are bidirectional means that, like grammar rules, they can be used for both analysis and synthesis. The fact that overlays are completely declarative gives them a firm semantic basis and makes it easier to reason about them.

Figure 8 is an example of the graphical notation for an overlay. The name of the overlay is *bump-and-update-as-push*. In general, an overlay diagram has a plan diagram on each side with a set of hooked lines, called *correspondences*, between them. The plan diagram on the left side defines the implementation, which is the domain of the mapping defined by the overlay; the plan diagram on the right side defines the specification, which is the range of the mapping; the correspondences define the mapping.

In this example, the right side of the overlay is a degenerate plan diagram with only a single box: the specification being implemented in this overlay is the *push* operation on a *list*. (List is a data abstraction having a *head* of any type and a *tail*, which is a list or empty.) The diagram shows that push has two inputs, the *old* list and the *input* (of any type), and one output, the *new* list. The postcondition of push (logical annotations are usually not shown in diagrams) specifies that the head of the new list is equal to the input and that the tail of the new list is equal to the old list.

The left side of this overlay is a plan diagram representing a clichéd combination of operations on an *indexed sequence* (a *base* sequence with an associated *index* integer between zero and the length of the base), in which the index is decremented and a new term is stored. The name of this plan is *bump-and-update*. It has four roles: *old* (an indexed sequence), *bump* (an operation that adds one), *update* (an operation that stores a new term in a sequence), and *new* (an indexed sequence).

There are three correspondences in this overlay. Two of these correspondences are annotated with the name of another overlay called *indexed-sequence-as-list*. This means that the old indexed sequence of bump-and-update, viewed as a list according to indexed-sequence-as-list, corresponds to the old input of push, and similarly for the new roles. Indexed-sequence-as-list is a data abstraction function defined as follows: the head of the list corresponds to the term of the base indexed by the index; the tail of the list is recursively defined as the list implemented by the indexed sequence with the same base and one minus the index; the empty list corresponds to the indexed sequence with index zero. The third correspondence in the diagram indicates that the input to the update operation in bump-and-update (the input for the new term—the other two inputs are the sequence and the index) corresponds to the object being pushed onto the list.

In order to give plan diagrams a precise definition, each aspect of plan diagrams is defined in terms of a version of a situational calculus. Manna and Waldinger (1980b) have used a situational calculus in a similar way in order to specify certain problematic features of programming languages.

Since the situational calculus is essentially just predicate calculus with some conventions applied, a plan diagram can be viewed as the abbreviation for a set of logical assertions. Given that the logical annotation in a plan also consists of predicate calculus assertions and overlays are just mappings between plans, this implies that everything in a plan can be reduced to a set of logical assertions.

5.3 KBEmacs

KBEmacs (for Knowledge-Based Editor in Emacs) is a prototype of a part of the Programmer's Apprentice completed several years ago to demonstrate the utility of inspection methods and the Plan Calculus in the implementation phase of the software process.

Figure 9 shows the architecture of KBEmacs. Two representations are maintained for the program being manipulated: program text (displayed to the programmer) and a plan (used to support KBEmacs' internal operation). At any moment, the programmer can modify either the text or the plan. If the text is modified, the *analyzer* is used to create a new plan. If the plan is modified, the *coder* is used to create new program text.

The program text can be modified using a standard program editor. In this case, the editor is Emacs (Stallman, 1981), which supports both text- and syntax-based program editing. The plan can be modified using *knowledge-based* commands supported by the *plan editor*. These commands rely on two kinds of knowledge. First, the commands can refer to clichés in the *cliché library*. Second, the plan editor itself contains a significant amount of procedurally embedded knowledge about manipulating and combining clichés.

While plans are crucial to the internal operation of KBEmacs, program text is used as the primary user interface. This allows a programmer to interact with KBEmacs using a familiar programming language and without studying the Plan Calculus. Further, since KBEmacs produces ordinary source code, standard programming tools (e.g., the compiler) can be used without any kind of special interface.

KBEmacs' interface unifies text-, syntax-, and knowledge-based editing so that they are all conveniently accessed through the standard Emacs-style editor. The knowledge-based commands are supported as a pseudo-English extension of the standard editor's command set. The results of knowledge-based commands are visible to the programmer as alterations in the program text in the editor buffer. The effect is as if a human assistant were modifying the text under the programmer's direction. The programmer can fall back on direct text- and syntax-based editing at any time.

Figure 10 shows an example of using KBEmacs to implement a 55 line Ada program. (We have omitted the intermediate states of the editing session after each command. See Waters (1985) for the full scenario.)

The clichés used in this scenario are: *simple report* (which is discussed further below), *chain enumeration*, and *query user for key*. Note that *enumerator*, *main file key*, *title*, and *summary* are the names of roles in these clichés. Our library of algorithmic and data structure clichés is similar in scope and level to earlier machine-usable codifications, such as Green and Barstow (1978); however, as discussed above, we have developed an improved formal representation.

One way of thinking of KBEmacs is that it adds a new, higher level of editing commands to the existing text- and syntax-based commands of Emacs. Using KBEmacs, changes in the algorithmic structure of a program can be achieved by a single command, even when they correspond to widespread textual changes in the program.

Two important capabilities of KBEmacs that are not illustrated in Figure 10, but which appear in the full scenario, are the automatic generation of program documentation (i.e., explaining the program in terms of the clichés used), and programming-language independence. KBEmacs was originally constructed to operate on Lisp pro-

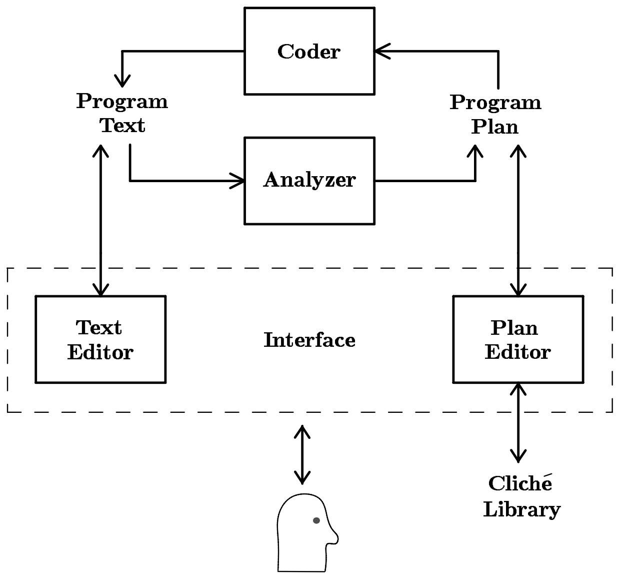

Figure 9. Architecture of KBEmacs.

grams; relatively little effort was required to extend it to operate on Ada programs as well.

Figure 11 illustrates a facility in KBEmacs for defining clichés using an extension of Ada syntax. KBEmacs automatically analyzes this text to extract the underlying plan for the cliché. Using KBEmacs commands, this plan may then be combined with the plans for other clichés to make new programs. Figure 11 defines the simple report cliché, which is used in the scenario of Figure 10. The only extensions to the syntax of Ada in cliché definitions are the new defining form `cliche ... is ...`, and the use of braces for role names and other annotation. Note, however, that this provides a more general form of parameterization than supported by the Ada package facility.

We envisage two kinds of cliché definition activity using the Programmer's Apprentice. First, this language-oriented interface makes it easy for a programmer to quickly define a cliché for his own (perhaps short-term) use. A second, much more intellectually demanding task, is to define a "suite" of clichés to be used by a large group of people over a long period of time.

Like most clichés, simple report includes some standard computation (e.g., the printing of the title page), some roles to be filled in (e.g., the title itself), and the data flow and control flow between them. This cliché has seven roles. The *file name* is the name of the file that will contain the report being produced. The *title* is printed on a title page and at the top of each succeeding page of the report (along with the page number). The *enumerator* enumerates the elements of some aggregate data structure. The *print item* is used to print out information about each of the enumerated elements. The *line limit* is used to determine when a page break should be inserted. The *column headings* are printed at the top of each page to explain the output of the print item. The *summary* prints out some summary information at the end of the report.

```
Define a simple_report procedure UNIT_REPAIR_REPORT.
Fill the enumerator with a chain_enumeration of UNITS and REPAIRS.
Fill the main_file_key with a query_user_for_key of UNITS.
Fill the title with ("Report of Repairs on Unit " & UNIT_KEY).
Remove the summary.
```

$$\Downarrow$$

```
with CALENDAR, FUNCTIONS, MAINTENANCE_FILES, TEXT_IO;
use CALENDAR, FUNCTIONS, MAINTENANCE_FILES, TEXT_IO;
procedure UNIT_REPAIR_REPORT is
   use DEFECT_IO, REPAIR_IO, UNIT_IO, INT_IO;
   CURRENT_DATE: constant STRING := FORMAT_DATE(CLOCK);
   DEFECT: DEFECT_TYPE;
   REPAIR: REPAIR_TYPE;
   REPAIR_INDEX: REPAIR_INDEX_TYPE;
   REPORT: TEXT_IO.FILE_TYPE;
   TITLE: STRING(1..33);
   UNIT: UNIT_TYPE;
   UNIT_KEY: UNIT_KEY_TYPE;
   procedure CLEAN_UP is
      begin
         SET_OUTPUT(STANDARD_OUTPUT);
         CLOSE(DEFECTS); CLOSE(REPAIRS); CLOSE(UNITS); CLOSE(REPORT);
      exception
         when STATUS_ERROR => return;
      end CLEAN_UP;
begin
   OPEN(DEFECTS, IN_FILE, DEFECTS_NAME); OPEN(REPAIRS, IN_FILE, REPAIRS_NAME);
   OPEN(UNITS, IN_FILE, UNITS_NAME); CREATE(REPORT, OUT_FILE, "report.txt");
   loop
      begin
         NEW_LINE; PUT("Enter UNIT Key: "); GET(UNIT_KEY);
         READ(UNITS, UNIT, UNIT_KEY);
         exit;
      exception
         when END_ERROR => PUT("Invalid UNIT Key"); NEW_LINE;
      end;
   end loop;
   TITLE := "Report of Repairs on Unit " & UNIT_KEY;
   SET_OUTPUT(REPORT);
   NEW_LINE(4); SET_COL(20); PUT(CURRENT_DATE);
   NEW_LINE(2); SET_COL(13); PUT(TITLE); NEW_LINE(60);
   READ(UNITS, UNIT, UNIT_KEY); REPAIR_INDEX := UNIT.REPAIR;
   while not NULL_INDEX(REPAIR_INDEX) loop
      READ(REPAIRS, REPAIR, REPAIR_INDEX);
      if LINE > 64 then
         NEW_PAGE; NEW_LINE; PUT("Page: "); PUT(INTEGER(PAGE-1), 3);
         SET_COL(13); PUT(TITLE); SET_COL(61); PUT(CURRENT_DATE); NEW_LINE(2);
         PUT("  Date    Defect   Description/Comment"); NEW_LINE(2);
      end if;
      READ(DEFECTS, DEFECT, REPAIR.DEFECT);
      PUT(FORMAT_DATE(REPAIR.DATE)); SET_COL(13); PUT(REPAIR.DEFECT);
      SET_COL(20); PUT(DEFECT.NAME); NEW_LINE;
      SET_COL(22); PUT(REPAIR.COMMENT); NEW_LINE;
      REPAIR_INDEX := REPAIR.NEXT;
   end loop;
   CLEAN_UP;
exception
   when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
      CLEAN_UP; PUT("Data Base Inconsistent");
   when others => CLEAN_UP; raise; end UNIT_REPAIR_REPORT;
```

Figure 10. A verbatim transcript from KBEmacs. Given the five commands above, it produced the 55 lines of Ada code shown.

```
with CALENDAR, FUNCTIONS, TEXT_IO; use CALENDAR, FUNCTIONS, TEXT_IO;
cliche SIMPLE_REPORT is
  primary roles ENUMERATOR, PRINT_ITEM, SUMMARY;
  described roles FILE_NAME, TITLE, ENUMERATOR, COLUMN_HEADINGS,
                  PRINT_ITEM, SUMMARY;
  comment "prints a report of {the input data of the enumerator}";
  constraints
    DEFAULT({the file_name}, "report.txt");
    DERIVED({the line_limit},
            66-SIZE_IN_LINES({the print_item})
              -SIZE_IN_LINES({the summary}));
    DEFAULT({the print_item},
            CORRESPONDING_PRINTING({the enumerator}));
    DEFAULT({the column_headings},
            CORRESPONDING_HEADINGS({the print_item}));
  end constraints;
  use INT_IO;
  CURRENT_DATE: constant STRING := FORMAT_DATE(CLOCK);
  DATA: {};
  REPORT: TEXT_IO.FILE_TYPE;
  TITLE: STRING(1..{});
  procedure CLEAN_UP is
  begin
    SET_OUTPUT(STANDARD_OUTPUT); CLOSE(REPORT);
  exception when STATUS_ERROR => return;
  end CLEAN_UP;
begin
  CREATE(REPORT, OUT_FILE, {the file_name});
  DATA := {the input data of the enumerator};
  SET_OUTPUT(REPORT);
  TITLE := {the title};
  NEW_LINE(4); SET_COL(20); PUT(CURRENT_DATE); NEW_LINE(2);
  SET_COL(13); PUT(TITLE); NEW_LINE(60);
  while not {the empty_test of the enumerator}(DATA) loop
    if LINE > {the line_limit} then
       NEW_PAGE; NEW_LINE; PUT("Page: "); PUT(INTEGER(PAGE-1), 3);
       SET_COL(13); PUT(TITLE);
       SET_COL(61); PUT(CURRENT_DATE); NEW_LINE(2);
       {the column_headings}({CURRENT_OUTPUT, modified});
    end if;
    {the print_item}({CURRENT_OUTPUT, modified},
                     {the accessor of the enumerator}(DATA));
    DATA := {the step of the enumerator}(DATA);
  end loop;
  {the summary}({CURRENT_OUTPUT, modified});
  CLEAN_UP;
exception
  when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
    CLEAN_UP; PUT("Data Base Inconsistent");
  when others => CLEAN_UP; raise;
end SIMPLE_REPORT;
```

Figure 11. An Ada presentation of the simple report cliché. This program text is analyzed by KBEmacs to extract the underlying plan, which is stored and later used in Figure 10.

The enumerator is a compound role with four sub-roles: the *input data*, the *empty test*, the *accessor*, and the *step*. These sub-roles can be filled individually, or they can be filled as a unit with an enumeration cliché (such as *chain enumeration* in Figure 10).

The simple report cliché also includes four constraints, which are specified at the beginning of the cliché definition. The first constraint specifies `"report.txt"` as the default name for the file containing the report. This name will be used unless the programmer specifies some other name.

The second constraint specifies that the line limit should be 65 minus the number of lines printed by the print item and the number of lines printed by the summary. Because the line limit role is computed by this constraint, the programmer never has to fill it explicitly; the role will automatically be updated if the print item or summary is changed. (For example, the line limit is computed to be 64 in Figure 10.)

The remaining two constraints provide default formats for printing items in the report and the column headings. If clichés have been defined for how to print a given type of object in a report and the corresponding headings, then the functions `CORRESPONDING_PRINTING` and `CORRESPONDING_HEADINGS` retrieve the appropriate clichés; otherwise, these functions are simple program generators that construct appropriate code based on the definitions of the types of objects involved. One of the deficiencies of KBEmacs is that the collection of functions used in constraints, such as `SIZE_IN_LINES`, `CORRESPONDING_PRINTING`, and `CORRESPONDING_HEADINGS`, is not easy for the programmer to extend.

Future work building on KBEmacs (Waters and Tan, 1991) is directed towards detecting more errors (using the automated reasoning facilities in Cake), automating some of the cliché selection, and generally raising the level of interaction toward higher level design decisions.

6 Conclusion

Basic research on automatic programming is very much like cancer research: A host of fundamental problems still remain to be solved. Therefore, it is highly unlikely that anyone will discover a "silver bullet" that will remove all obstacles to the rapid development of general-purpose automatic programming. However, researchers will continue to chip away at the problem from many directions.

6.1 Commercially Available Systems

Academic research on automatic programming has focused on developing techniques that can support broad-coverage, fully automatic programming. Unfortunately, while this research points in the direction of long term progress, it has not yet had very much impact on commercial systems.

Work in the commercial arena has focused on more modest goals and has been able to make significant steps toward automatic programming based on procedural methods. In particular, development has quickened over the last few years with the introduction of so-called Computer-Aided Software Engineering or CASE.

6.1.1 Data base query systems

Perhaps the greatest commercial automatic programming success story has been the development of data base query systems (e.g., Information Builders' Focus). These systems have limited capabilities and are not suitable for complex applications. However, they allow end users to retrieve information from a data base and produce customized reports without the help of programmers.

Within their narrow domain of applicability, data base query systems are both end-user oriented and fully automatic. In simple applications, these systems have completely taken over, making automatic programming an everyday reality.

6.1.2 Fourth-generation languages

Following the bottom-up approach to automatic programming, a number of commercial systems have been introduced that achieve a broader range of coverage than data base query systems. They do this by sacrificing end-user orientation. Most such systems offer a combination of special-purpose interfaces (such as screen painters and report generators) and a very high level language designed specifically for business data processing applications. Systems that execute their languages interpretively, such as Applied Data Research's Ideal and Software ag's Natural, are typically called fourth-generation languages.

Fourth-generation languages are used to some extent at perhaps ten thousand sites. However, though there is a great deal of enthusiasm about their potential, fourth-generation languages are very far from displacing Cobol. This is because they are relatively inefficient and because they cannot be used conveniently in conjunction with pre-existing applications.

6.1.3 Program generators

Program generators, such as Transform Logic's Transform and Pansophic Systems' Telon, are very similar to fourth-generation languages, except that they generate Cobol code rather than operating interpretively. In exchange for this increase in efficiency, program generators must settle for supporting a narrower range of features.

Program generators are used at approximately a thousand sites. Although more efficient than fourth-generation languages, their acceptance is limited by their narrower focus and by the difficulty of using them in conjunction with pre-existing code.

6.1.4 High-level design aids

Graphical tools, such as Index Technology's Excelerator, support the manipulation of high-level designs without being able to generate executable code. High-level design aids therefore exemplify the assistant approach to automatic programming, rather than the bottom-up approach.

Tools of this general type are used at several thousand sites and are rapidly becoming a standard part of the programming process. However, their acceptance is slow because they lack integration with other tools and because they leave code generation to the user.

6.1.5 Project management tools

Also in the spirit of the assistant approach to automatic programming, we should note the growing capabilities of project management tools. These tools provide relatively modest but significant support for managing the programming process. For example, products such as BIS Applied Systems' BIS/IPSE, and Imperial Software Technology's ISTAR provide facilities for breaking down a project into tasks and tracking their progress, for configuration and version control, and for the generation of various kinds of documentation and management reports.

If in-house tools are counted, programming management aids are rapidly on the way to becoming the norm rather than the exception in large projects. Assuming that automatic programming is unlikely to make the problems of managing cooperative work disappear, the need for such tools will continue.

6.1.6 Very high level prototyping languages

The one place where academic research has had a significant impact on commercial systems is in very high level prototyping languages. These languages represent a compromise between desires and reality. While researchers would like to create extremely high level languages that could be compiled into efficient code, it is not yet possible—even with significant sacrifices in the language—to create production quality code. The current status of general-purpose, very high level prototyping languages is typified by Reasoning System's Refine (Abraido-Fandino, 1987), which is based on research initiated at Stanford. Prolog (Cohen, 1985), which is based on logic programming research at Imperial College, is also being used as a very high level prototyping language.

The exact extent of very high level prototyping languages usage is not clear. However, it probably does not exceed a hundred sites. Acceptance of this approach is currently limited by the fact that rapid prototyping as a methodology is far from universally accepted.

6.2 On the Horizon

Over the next several years, progress toward automatic programming will almost certainly follow the course set by currently available systems. Although conditions seem ready for relatively rapid progress in CASE tools, radical breakthroughs seem unlikely. Rapid progress is possible primarily in the ways in which currently available systems are used.

6.2.1 Technological advances

The quality of commercially available programming tools should improve markedly over the next several years. In particular, high-level design aids (e.g., Texas Instruments' IEF) will be extended to generate executable code in many situations. Fourth generation languages and program generators will add support for somewhat higher level constructs and somewhat less narrow domains of applicability. In addition, there will be a general trend toward greater integration of programming tools. With any luck, these incremental improvements should be enough to promote most of these tools from experimental usage to full-scale acceptance.

The developers of very high level prototyping languages are strongly committed to increasing the efficiency of the code produced. Some of the inefficiency is more or less incidental and will undoubtedly be eliminated. However, other problems are intrinsic to the approach: The whole point of very high level languages is to write a program using algorithms oriented toward clarity rather than efficiency. Since clear algorithms are often very inefficient, efficiency often requires radical changes. Unfortunately, no one knows how to identify such changes automatically or how to take advice on the subject effectively.

To date, essentially all of the commercialization of automatic programming research has been via the very high level language approach. However, in the near future, we will begin to see the first commercialization of research on the assistant approach.

Rapidly decreasing prices for workstation and data base hardware provide an important opportunity. Soon, a threshold will be reached where it will be practical to capture on-line all of the intermediate work products of the programming process, whether produced manually or automatically. Besides being beneficial in its own right, this will drive further automation.

6.2.2 Management changes

Progress in any kind of automation is always obstructed by management problems as much as by technological hurdles. At least four major changes must take place at the management level if the potential of automatic programming is to be realized.

First, we must recognize that the capitalization for programming needs to be increased. In most organizations, a dollar spent on additional computer hardware or programming tools will bring significantly more benefits than a dollar spent on additional programmers. (Studies have shown that significant productivity gains can be obtained merely by giving programmers offices with doors!)

Second, given that the heart of automatic programming is reuse, the economic incentives in software development and acquisition need to be revised to foster reuse. Under current contracting practices, there is often an economic incentive *against* software reuse and the production of easily-maintainable software. Policies whereby contractors would increase their profit by reusing software developed by someone else—or were paid extra if they produced something that someone else reused—would be steps in the right direction. It would also be a good idea to tie some part of profit to the long-term costs of the delivered software.

Third, management must recognize that the only way to reduce the lifetime costs of software is to spend more supporting the early parts of the process—requirements definition, specification, and design. For example, people often talk about software reuse as if it were some miraculous way to reuse code that has already been written. In fact, there is no way to reuse software unless it is carefully designed to be reusable. This pays big dividends, but it requires significant "up front" expenditures.

Finally, as with all automation, the real promise of automatic programming is not just in automating what is done now but in completely changing the way things are done. In the case of office automation, for example, it pays to redesign the whole information flow in the office, rather than put the same old paper forms into an electronic medium. With programming, this means reexamining the traditional model of the software lifecycle, which is beginning to happen with the increasing acceptance of prototyping. It also means breaking down the conventional distinctions between languages, environments, and interfaces, which is occurring in the form of graphical interfaces, and object-oriented programming.

References

Abraido-Fandino, L. M. (1987). An overview of Refine 2.0. In *2nd Int. Symp. on Knowledge Engineering–Software Engineering*, Madrid, Spain.

Ada (1983). *Military Standard Ada Programming Lanuage*. Dept. of Defense, U.S. Govt. Printing Office. ANSI/MIL-STD-1815A-1983.

Andreae, P. M. (1985). Justified generalization: Acquiring procedures from examples. Technical Report 834, MIT Artificial Intelligence Lab. PhD thesis.

Babb, R. G. (1985). Workshop on models and languages for software specification and design. *IEEE Computer*. 18(3), 103–108.

Balzer, R. M. (1985). A 15 year perspective on automatic programming. *IEEE Trans. Software Engineering*. 11(11), 1257–1267.

Barstow, D. R. (1984). A perspective on automatic programming. *AI Magazine*. 5(1), 5–27. Reprinted in C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

Basu, S. K. and Misra, J. (1976). Some classes of naturally provable programs. In *2nd Int. Conf. on Software Engineering*, San Francisco, CA.

Bauer, M. A. (1979). Programming by examples. *Artificial Intelligence*. 12, 1–21. Reprinted in C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

Biermann, A. (1972). On the inference of Turing machines from sample computations. *Artificial Intelligence*. 3(3).

Biermann, A. (1976). Approaches to automatic programming. In Yovits, M. C., editor, *Advances in Computers, Volume 15*, pages 1–63. Academic Press, Boston, MA.

Biermann, A. and Krishnaswamy, R. (1976). Constructing programs from example computations. *IEEE Trans. Software Engineering*. 2(3).

Cheatham, T. E. (1984). Reusability through program transformation. *IEEE Trans. Software Engineering*. 10(5), 589–595. Reprinted in C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

Cheng, T. T., Lock, E. D., and Prywes, N. S. (1982). Use of very high level languages and program generation by management professionals. *IEEE Trans. Software Engineering*. 10(5), 552–563.

Cohen, J. (1985). Describing Prolog by its interpretation and compilation. *Comm. ACM*. 28(12), 1311–1324.

Feather, M. S. and London, P. E. (1982). Implementing specification freedoms. *Science of Computer Programming*. 2, 91–131.

Gerhart, S. L. (1975). Knowledge about programs: A model and case study. In *Proc. Int. Conf. on Reliable Software*, pages 88–95. Published as ACM SIGPLAN Notices, 10(6).

Green, C. and Barstow, D. R. (1978). On program synthesis knowledge. *Artificial Intelligence*. 10(3), 241–279. Reprinted in C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

Hardy, S. (1975). Synthesis of LISP functions from examples. In *Proc. 4th Int. Joint Conf. Artificial Intelligence*, pages 240–245, Tblisi, Georgia, USSR.

Hedrick, C. L. (1976). Learning production systems from examples. *Artificial Intelligence*. 7, 21–49.

Heidorn, G. E. (1976). Automatic programming through natural language dialogue: A survey. *IBM J. Research & Development*. 20(4), 302–313. Reprinted in C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

Ianov, Y. I. (1960). The logical schemes of algorithms. In *Problems of Cybernetics*, pages 82–140. Pergamon Press. Vol. 1, English translation.

Lieberman, H. and Hewitt, C. (1980). A session with TINKER: Interleaving program testing with program design. In *Proc. Conf. on LISP*, Stanford, CA.

Liskov, B. H. and Guttag, J. (1986). *Abstraction and Specification in Program Development*. McGraw-Hill, New York, NY.

Low, J. R. (1978). Automatic data structure selection: An example and overview. *Comm. ACM*. 21(5), 376–384.

Manna, Z. (1974). *Mathematic Theory of Computation*. McGraw-Hill, New York, NY.

Manna, Z. and Waldinger, R. (1980a). A deductive approach to program synthesis. *ACM Trans. Programming Languages and Systems*. 2(1), 90–121. Reprinted in C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

Manna, Z. and Waldinger, R. (1980b). Problematic features of programming languages: A situational-calculus approach; Part I: Assignment statements. Technical report, Stanford Univ. & The Weizmann Institute & The Artificial Intelligence Center.

Neighbors, J. M. (1984). The Draco approach to constructing software from reusable components. *IEEE Trans. Software Engineering*. 10(5), 564–574. Reprinted in C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

Reubenstein, H. B. and Waters, R. C. (1991). The Requirements Apprentice: Automated assistance for requirements acquisition. *IEEE Trans. Software Engineering*. 17(3), 226–240.

Rich, C. and Feldman, Y. A. (1992). Seven layers of knowledge representation and reasoning in support of software development. *IEEE Trans. Software Engineering*. 18(6), 451–469. Special Issue on Knowledge Representation and Reasoning in Software Development. Also published as MERL Technical Report 92-01.

Rich, C. and Waters, R. C. (1988). The Programmer's Apprentice: A research overview. *IEEE Computer*. 21(11), 10–25. Reprinted in D. Partridge, editor, *Artificial Intelligence and Software Engineering*, pages 155–182, Ablex, Norwood, NJ, 1991, and in P. H. Winston with S. A. Shellard, editors, *Artificial Intelligence at MIT: Expanding Frontiers*, pages 166–195, MIT Press, Cambridge, MA, 1990.

Rich, C. and Waters, R. C. (1990). *The Programmer's Apprentice*. Addison-Wesley, Reading, MA and ACM Press, Baltimore, MD.

Rowe, L. A. and Tonge, F. M. (1978). Automating the selection of implementation structures. *IEEE Trans. Software Engineering.* 4(6), 494–506. Reprinted in C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

Ruth, G. R. (1978). Protosystem I: An automatic programming system prototype. In *Proc. AFIPS National Computer Conf.*, pages 675–681, Anaheim, CA. Reprinted in C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

Schwartz, J. T., Dewar, R. B. K., Dubinsky, E., and Schonberg, E. (1986). *Programming with Sets: An Introduction to SETL.* Springer-Verlag, New York, NY.

Shaw, D., Swartout, W., and Green, C. (1975). Inferring LISP programs from examples. In *Proc. 4th Int. Joint Conf. Artificial Intelligence*, pages 260–267, Tblisi, Georgia, USSR.

Siklossy, L. and Sykes, D. (1975). Automatic program synthesis from example problems. In *Proc. 4th Int. Joint Conf. Artificial Intelligence*, pages 268–273, Tblisi, Georgia, USSR.

Smith, D. R. (1984). The synthesis of LISP programs from examples: A survey. In Biermann et al, A. W., editor, *Automatic Program Construction Techniques*, pages 307–324. Macmillan, New York, NY.

Stallman, R. M. (1981). EMACS: The extensible, customizable self-documenting display editor. In *Proc. ACM SIGPLAN/SIGOA Symp. on Text Manipulation*, Portland, OR.

Summers, P. D. (1977). A methodology for LISP program construction from examples. *J. ACM.* 24(1), 161–175. Reprinted in C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

Swartout, W. (1983). The GIST Behavior Explainer. In *Proc. 3rd National Conf. on Artificial Intelligence*, pages 402–407, Washington, DC.

Waters, R. C. (1985). The Programmer's Apprentice: A session with KBEmacs. *IEEE Trans. Software Engineering.* 11(11), 1296–1320. Reprinted in C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

Waters, R. C. and Tan, Y. M. (1991). Toward a Design Apprentice: Supporting reuse and evolution in software design. *ACM SIGSOFT Software Engineering Notes.* 16(2), 33–44.

Wills, L. M. (1990). Automated program recognition: A feasibility demonstration. *Artificial Intelligence*. 45(1–2), 113–172.

Wills, L. M. (1992). Automated program recognition by graph parsing. Technical Report 1385, MIT Artificial Intelligence Lab. PhD thesis.

Wirth, N. (1973). *Systematic Programming, An Introduction*. Prentice Hall, Englewood Cliffs, NJ.

Wulf, W. A., London, R. L., and Shaw, M. (1976). An introduction to the construction and verification of Alphard programs. *IEEE Trans. Software Engineering*. 2(4), 253–265.