Automatic Programming: Myths and Prospects

Charles Rich and Richard C. Waters
Massachusetts Institute of Technology

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 (see the sidebar on its status in 1958). On the other hand, current expectations regarding its potential 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 in commercially available systems.

The "cocktail party" description of automatic programming is unachievable because it is based on faulty assumptions; nevertheless, we will see continued evolutionary progress.

Automatic programming myths and realities

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.

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 for them 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 progression from a description that can be understood only in the context of the domain to a description that can be understood without relying on auxiliary knowledge (see the sidebar on agents in the programming process). There is no point at which someone who knows nothing about programming communicates directly with someone who knows nothing about the application domain.

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 the speaker does and can therefore understand the words and fill in what is left out. (For a detailed discussion of the necessity of domain knowledge in automatic programming, see Barstow.[1])

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 expert in every application domain. Unfortunately, artificial intelligence is nowhere near supporting this superhuman level of performance.

Reality: Given the pragmatic impossibility of simultaneously supporting all three features, 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:

(1) *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.

(2) *Narrow domain.* This approach sacrifices being general purpose. Focusing 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.

(3) *Assistant.* This approach sacrifices full automation. Instead, it seeks to assist in various aspects of programming. 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 the level of assistance provided by individual tools.

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 work only when both parties make

Automatic programming in 1958

The chart below lists the "automatic programming systems" known to ACM's editors as of April 1958. It specifies the computer each system runs on and indicates whether comprehensive documentation for the system was available in a library maintained by the ACM. Most of the systems listed are what we now call assemblers; a few (notably Fortran for the IBM 704) are compilers.

The principal lesson provided by this chart is that the term "automatic programming" has always been relative rather than absolute. In 1988, no one would call any of these systems automatic programming. In 1958, however, 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 operate computers without having to hire a programmer.

The chart has been reprinted with permission from the ACM (*Communications of the ACM*, Vol. 1, No. 4, April 1958, page 8).

AUTOMATIC PROGRAMMING SYSTEMS

Computer	In library	Do not have
704	AFAC CAGE FORTRAN NYAP PACT IA REG-SYMBOLIC SAP	ADES FORC KOMPILER 3
701	DUAL-607 FLOP JCS-13 KOMPILER 2 QUICK SHACO SPEEDCODING 3	BACAIC DOUGLAS GEPURS LT-2 PACT I QUEASY SEESAW SO 2 SPEEDEX
705	ACOM AUTOCODER ELI PRINT I SOHIO SYMB. ASSEMBLY	FAIR
702	AUTOCODER ASSEMBLY SCRIPT	
1103–1103A	CHIP FAP FLIP-SPUR MISHAP RAWOOP-SNAP UNICODE USE	COMPILER I TRANS-USE
MIDAC	EASIAC MAGIC	
650	ADES II BELL BELL L2, L3 DRUCO I EASE II ELI FOR TRANSIT IT RELATIVE SIR SOAP I SOAP II	BACAIC BALITAC ESCAPE FLAIR MITILAC OMNICODE SPEEDCODING SPUR
UNIVAC I, II	A2 ARITHMATIC (A3) GP MATHMATIC (AT3) NYU, OMNIFAX	A0, A1 FLOWMATIC (B–O) BIOR MJS RELCODE SHORTCODE X-1
D'TRON 201 204 205	UGLIAC	APS DATACODE I DUMBO IT SAC STAR
UDEC III		UDECIN-I UDECOM-3 INTERCOM
G-15		ALGEBRAIC
WHIRLWIND	COMPREHENSIVE SUMMER SESSION	
FERUT	TRANSCODE	
JOHNNIAC	EASY FOX	

Agents in the programming process

Suppose a large company needs a new accounting system. This figure shows the principal agents that typically would 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 toward the bottom.

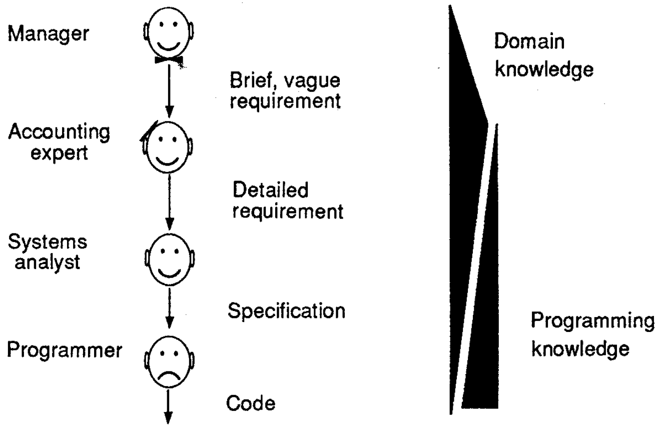
The manager at the top of the figure quite likely has only a rudimentary knowledge of accounting. The manager's job is to identify a need and initiate the programming process by creating a brief, vague requirement. Use of the term "vague" here highlights the fact that the only way this initial requirement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

The next agent in the process is an accounting expert, whose 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 for evaluation 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 systems analyst, whose job is to define the basic architecture of the program and 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 systems analyst must have a considerable understanding of accounting in addition to an extensive knowledge of programming.

The final agent is a programmer, who must produce code in a high-level language on the basis of 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.



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, there is no practical limit to what must 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 inserting the bank card the customer should enter a password, select an account, and then select an amount of cash which the machine then dispenses. However, this is nowhere near complete.

To start with, a lot of details are missing regarding the user interface: What kinds of directions are displayed to the customer? How is the customer to select among 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 prevent possible tampering with bank cards?

Looking 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.

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

Finally, the end user also cares about the cost of implementing the program and how long implementation will take. This implies a need for trade-offs, particularly when it comes to the last few issues mentioned above. Thus, it is very difficult—if not impossible—to make complete statements about these issues.

Reality: At best, requirements are only approximations. Instead of serving as a defensive measure between adversaries, requirements should serve as a tool for communication between colleagues. Assuming that the implementer 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. 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.

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 significant feedback to flesh out their ideas and determine the exact 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.

Reality: Programming is an iterative process featuring continual dialogue between end user and programmer. The desired requirement evolves on the basis 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 have 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.

Automatic programming systems of the future will be more like vacuum cleaners than like self-cleaning ovens.

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. Undoubtedly these new inputs will 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.

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 spreadsheet over time differs very little from maintaining a program. The only real difference is that a spreadsheet is a concise domain-specific interface that makes it remarkably easy to write certain kinds of programs and startlingly hard to write other kinds of programs.

Myth: There will be no more programming in the large. Even if we accept 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 used to attack applications that are enormous rather than merely huge.

Reality: We are not likely 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 the other accoutrements of cooperative work and programming in the large.

A down-to-earth perspective. 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.

Our discussion of the fundamental technical issues in automatic programming will be divided according to three 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?

We will also review the most recent trends in currently available automated programming tools. Under the rubric of computer-aided software engineering, or CASE, these tools are following the bottom-up, narrow-domain, and assistant approaches to automatic programming.

Finally, we offer ideas on the kinds of tools that will soon be available. CASE tools in particular are poised to emerge from a somewhat rocky adolescence into maturity. In this regard, we observe that the path toward automatic programming is impeded as much by the need for managerial change as by the need for technical advances.

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. The range of possibilities is illustrated in the accompanying sidebar on input languages and is further discussed below.

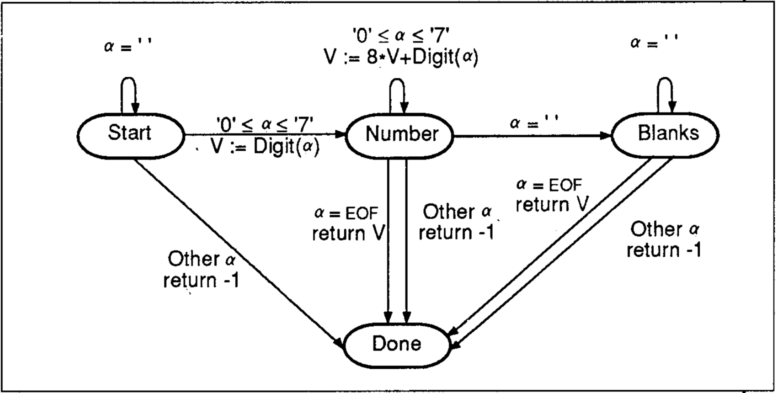
Natural language. Because they are familiar, natural languages such as English

Potential inputs to an automatic programming system

The figures in this sidebar illustrate the wide variety of inputs an automatic programming system might support. Each example is a specification for a program that determines the value of an octal number represented in a string. A Pascal implementation is included that illustrates what the output of an automatic programming system might look like. Note that no single example can illustrate all of the important issues in selecting an input medium.

The function EvalOctal is a recognizer that determines whether 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.

A natural language (English).



A special-purpose language (state transition diagrams).

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

Examples (input/output pairs).

$$
\begin{aligned}
\textit{input:}\ & S \\
\textit{precondition:}\ & \mathrm{String}(S) \\
\textit{output:}\ & V \\
\textit{postcondition:}\ & \mathrm{Integer}(V) \wedge (\mathrm{Valid}(S) \rightarrow \mathrm{Value}(S, V)) \wedge (\neg\mathrm{Valid}(S) \rightarrow V=-1)
\end{aligned}
$$

where:

$$
\begin{aligned}
\mathrm{Valid}(S) \equiv\ & (\forall i\ 1 \le i \le |S| \rightarrow S(i) \in \{\text{' ','1','2','3','4','5','6','7'}\}) \wedge \\
& (\exists i\ 1 \le i \le |S| \rightarrow S(i) \ne \text{' '}) \wedge \\
& (\neg\exists ijk\ 1 \le i < j < k \le |S| \wedge S(i) \ne \text{' '} \wedge S(j) = \text{' '} \wedge S(k) \ne \text{' '}) \\
\mathrm{Value}(S, V) \equiv\ & \forall ij\ (1 \le i < j \le |S| \wedge S(i) \ne \text{' '} \wedge S(j) \ne \text{' '} \wedge (j = |S| \vee S(j+1) = \text{' '})) \\
& \rightarrow (\mathrm{Digit}(S(i)) = \mathrm{Div}(\mathrm{Rem}(V, 8^{j-i-1}), 8^{j-i-2}) \\
& \wedge ((i = 1 \vee S(i-1) = \text{' '}) \rightarrow V < 8^{j-i}))
\end{aligned}
$$

A logical formalism (predicate calculus).

```
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;
```

A very high level language (SETL).

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

A high-level language (Pascal).

are an attractive choice for communication between end users and an automatic programming system. Three features that make natural language attractive are vocabulary, informality, and syntax. As discussed later, the existing vocabulary of thousands of predefined words contributes most to making natural language an efficient communication medium.

Informality (for example, the possibility of a statement's being ambiguous, incomplete, contradictory, and/or inaccurate) is also very important. In fact, it is essential to a powerful strategy for dealing with complexity: Start with an almost-right description and incrementally modify it until it is acceptable. An interesting research direction is the design of artificial languages that intentionally allow informality.

Syntax is the least important feature of natural language. 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 way beyond the current capabilities 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.

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 (mathematical formulas and circuit diagrams, for instance) that experts routinely use in preference to natural language.

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 the 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 by simply making a picture of how the screen should look. Then a code generator automatically writes the code to drive the terminal and access the database.

Unfortunately, special-purpose languages have a fundamental problem: They are essentially useless outside 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 situations where the special-purpose language is appropriate. Even when multiple special-purpose input languages are supported,[2] the user can only combine the languages in simple ways. Much more work is necessary before special-purpose languages can reach their full potential as part of the interface to general-purpose systems.

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. The appeal of this approach is that nonprogrammers 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.

Unfortunately, except for toy problems, no one has been able to make an example-based automatic programming system work, and there is reason to believe this failure is fundamental. It is trivial, but useless, to construct a program that duplicates a particular set of examples and does nothing else. What is desired is a program that operates on whole classes of input data in a manner "analogous to" the examples. However, experience has shown that no matter how many examples are provided, there is no way to ensure that the generalization derived will be correct—without placing severe constraints on the domain of possible generalizations.

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 the next section). Second, complex logical formulas are notoriously difficult for most people to write and understand.

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[3] guarantees executability of logical descriptions by placing strong restrictions on the form of expressions.

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 data types, 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.[4] More recent very high level languages, such as Refine[5] and Gist,[6] have added other features—for example, constraints and nondeterminism.

Other communication issues. Beyond the topics discussed above, the following three general issues apply to any communication medium.

First, a medium should be *wide-spectrum*. The user should be able to specify everything from very abstract properties to low-level implementation advice. This is necessary (at least for the foreseeable future) because automatic programming systems cannot operate without getting a certain amount of advice at all levels. It is desirable for wide-spectrum communication to be supported in a single coherent formalism. However, in addition to a general-purpose wide-spectrum language, the ideal automatic programming system would support a number of special-purpose languages.

Second, because of programming's inherently iterative nature, a medium must be able to support a dialogue between the user and the automatic programming system. Therefore, serious attention must be paid to the language the system is going to use when speaking to the user. In addition, the input language must be capable of expressing "metalevel" information, that is, information about changes to the state of knowledge. One can imagine how natural language would serve well as a dialogue medium; however, restricted notations, such as very high level languages, are clearly not sufficient by themselves.

Third, a medium should come with a large vocabulary of predefined terms so that the system can converse with the user at a suitably high level. Given a choice, most users would prefer 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.

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 a basis for such systems are procedural methods, deductive methods, transformational methods, and inspection methods.

Procedural methods. To date, the most successful approach has been to simply write a special-purpose program that gets the right results. For example, most current compilers and program generators are essentially procedural in nature, although a few 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 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.

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.[7] In principle, any method of automated deduction—resolution, natural deduction, reasoning about anonymous individuals—can be used to support automatic programming.

Like engineers in other disciplines, programmers think mostly in terms of clichés.

Unfortunately, in practice none of these methods is yet able to prove 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.

Transformational methods. Transformational implementation systems[8] (for example, TI[6] and PDS[9]) dominate current 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 should be applied. If the applicability conditions are satisfied, the action is 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.

Two basic kinds of transformations exist. Some transformations replace specification-like constructs (for example, quantification over a set) with conventional constructs (for example, iteration over a list). These transformations encode knowledge of how to implement algorithms and data structures. Other transformations perform rearrangements and optimizations (for example, 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 capable of expressing both the user's input and the desired result. 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 condition is satisfied (for example, until there are no more very high level constructs).

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 various compilers and other systems.

An interesting aspect of transformation sequences is that they usually contain a small percentage of key steps (typically making decisions about how to implement abstractions) interleaved with many small, less intuitive steps that set things up and move things around. Current research on transformational methods is directed toward automating the many small steps while seeking user advice on the key steps.

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

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.

Benefits of automatic programming can be traced almost exclusively to reuse.

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 for communicating explanations and advice between the system and the user.

Following the assistant approach, the Programmer's Apprentice project[10,11] has demonstrated several aspects of inspection methods. For example, given a library of clichés, the system can automatically analyze a program to identify the algorithms used. In addition, programs can be constructed by combining user-selected clichés. Current research is directed toward automatic selection of some of the clichés to use.

In human programming, inspection methods are the most effective approach, whenever applicable. However, since inspection methods are ultimately based on experience, they apply 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.

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:

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.

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

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.

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.

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 restricts the ways in which the programs can interact. However, it is independent of the details of the computations performed by the 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, mas-

Programming-knowledge representations

This diagram traces the inheritance of ideas among the major approaches that have been used to represent programming knowledge. The two oldest approaches are subroutines and the encoding of knowledge in procedures that "write the right code." Subroutines are very limited in expressive power but are easy to combine. In contrast, procedural encoding has unlimited expressive power but makes it very hard to combine clichés.

Program schemas extend the expressive power of subroutines, while macros are essentially a restricted and more tractable form of procedural encoding. Flowcharts and flowchart schemas (especially those that include dataflow as well as control flow) introduce the idea of programming-language independence.

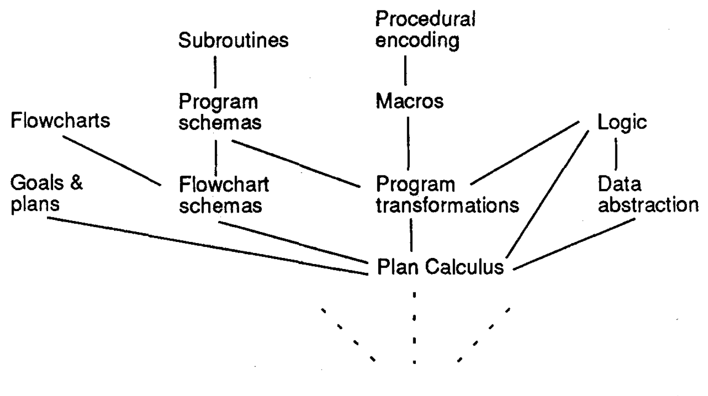
Logic can express even the most diffuse clichés in a declarative fashion. However, because of the weakness and inefficiency of current automatic theorem provers, pure logic is not sufficiently machine manipulable to serve as the sole representation for programming clichés. Data abstraction approaches combine program schemas (to specify abstract operations) with logic (to specify data structure invariants).

Goal and plan representations are used to explain the structure of programs at a deeper level than source text. This information is essential if an automatic programming system is to explain its actions.

Program transformations[8] incorporate ideas from program schemas, macros, and logic. As discussed in the subsection on transformational methods, a transformation has three parts: a pattern (which is essentially a program schema), a set of logical applicability conditions, and an action (which is essentially a macro).

The Plan Calculus[11] combines ideas from many of the representations described above. It achieves programming-language independence through the use of dataflow 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. And, it includes the concept of language-independent, bidirectional program transformations, which link pairs of flowchart schemas.

None of the representations above is completely satisfactory. If automatic programming systems are to continue to improve, representations must be developed that are both easier to manipulate and capable of representing aspects of programming knowledge (such as efficiency information) that are not readily captured by any current formalism.



ter 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:

Expressiveness—The representation must be able to express as many different kinds of clichés as possible.

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.

Semantic soundness—The representation must be based on a mathematical foundation that allows correctness conditions to be stated.

Machine manipulability—It must be possible to manipulate the representation effectively using computer tools.

Programming-language independence—The representation should not be tied to the syntax of any particular programming language.

In light of this "wish list," an accompanying sidebar discusses the various representations developed to date for programming clichés.

Commercially available systems

Academic research in automatic programming has focused on developing techniques that can support broad-coverage, fully automatic programming. Unfortunately, while this research points toward 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.

Database query systems. Perhaps the greatest commercial automatic programming success story has been the development of database query systems (for example, 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 database and produce customized reports without the help of programmers.

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

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 database 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 A.G.'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 great enthusiasm about their potential, fourth-generation languages are far from displacing Cobol. This is because they are relatively inefficient and cannot be used conveniently in conjunction with preexisting applications.

Program generators. Program generators, such as Transform Logic's Transform and Pansophic Systems' Telon, are very similar to fourth-generation languages except that instead of operating interpretively, they generate Cobol code. 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 preexisting code.

High-level design aids. Graphical tools, such as Index Technology's Excelerator, that support high-level software design methodologies take a different tack. These systems 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 they leave code generation to the user.

Project management tools. While considering the assistant approach to automatic programming, we should also point out 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, both 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.

Very high level prototyping languages. The one place where academic research has significantly affected 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 Systems' Refine,[5] which is based on research initiated at Stanford University. Prolog,[3] 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 language 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.

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 point to 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.

Technological advances. The quality of commercially available programming tools should improve markedly in the next few years. In particular, high-level design aids (for example, Texas Instruments' IEF) will be extended to generate executable code in many situations. Fourth-generation languages and program generators will add support for slightly 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 use to full-scale acceptance.

The developers of very high level prototyping languages, such as Refine, are strongly committed to increasing the efficiency of the code produced. Some 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, and 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 commercialization of automatic programming research has been via the very high level language approach. However, we will soon begin to see the first commercialization of research on the assistant approach. For example, Bachman Information Systems is developing a programming assistant product based in part on research at MIT.

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

Basic research on automatic programming is very much like cancer research: A host of fundamental problems 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.

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 occur at the management level if the potential of automatic programming is to be realized.

First, we must recognize that 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, 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.[12] Policies whereby contractors would increase their profit by reusing software developed by others—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 life cycle, which is beginning to happen with the increasing acceptance of prototyping. It also means breaking down conventional distinctions between languages, environments, and interfaces, which is occurring in the form of graphical interfaces and object-oriented programming.

Automatic programming in the form of compilers for high-level languages became available in the late 1950s. By the late 1960s, it was clear that the next logical step was to move up to very high level languages. However, this step turned out to be much more difficult than expected, and progress on the bottom-up approach to automatic programming was essentially stalled during the 1970s.

The main focus of work on automatic programming in the 1970s switched to the narrow-domain approach. Since then, a variety of systems, such as database query languages, have been constructed that deliver end-user-oriented, fully automatic programming in small domains.

In the 1980s, interest has returned to the bottom-up approach. This has led to the appearance of very high level prototyping languages. In addition, we have seen the arrival of fourth-generation languages and program generators that are more narrow in their focus as well as more efficient. The 1980s have also seen increased interest in the assistant approach to automatic programming in the guise of high-level design aids and other advanced programming tools.

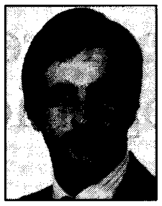
A further look into the future reveals no sign of the cocktail party version of automatic programming. However, there will be significant evolutionary progress. With luck, we will be saying much the same thing about automatic programming in 1998 that we said in 1958—that it has improved programmer productivity dramatically and has further reduced the distinction between programmers and end users. □

References

1. D.R. Barstow, "A Perspective on Automatic Programming," *AI Magazine*, Vol. 5, No. 1, Spring 1984, pp. 5-27.
2. J.M. Neighbors, "The Draco Approach to Constructing Software from Reusable Components," *IEEE Trans. Software Eng.*, Vol. 10, No. 5, Sept. 1984, pp. 564-574.
3. J. Cohen, "Describing Prolog by Its Interpretation and Compilation," *Comm. ACM*, Vol. 28, No. 12, Dec. 1985, pp. 1311-1324.
4. J.T. Schwartz et al., *Programming with Sets: An Introduction to SETL*, Springer-Verlag, New York, 1986.
5. L.M. Abraido-Fandiño, "An Overview of Refine 2.0," *Proc. Second Int'l Symp. Knowledge Eng.-Software Eng.*, Madrid, Apr. 1987.
6. R.M. Balzer, "A 15-Year Perspective on Automatic Programming," *IEEE Trans. Software Eng.*, Vol. 11, No. 11, Nov. 1985, pp. 1257-1267 (special issue on artificial intelligence and software engineering).

7. Z. Manna and R. Waldinger, "A Deductive Approach to Program Synthesis," *ACM Trans. Programming Languages and Systems*, Vol. 2, No. 1, Jan. 1980, pp. 90-121.
8. H. Partsch and T. Steinbrüggen, "Program Transformation Systems," *ACM Computing Surveys*, Vol. 15, No. 3, Sept. 1983, pp. 199-236.
9. T.E. Cheatham, "Reusability Through Program Transformation," *IEEE Trans. Software Eng.*, Vol. 19, No. 5, Sept. 1984, pp. 589-595.
10. R.C. Waters, "The Programmer's Apprentice: A Session with KBEmacs," *IEEE Trans. Software Eng.*, Vol. 11, No. 11, Nov. 1985, pp. 1296-1320 (special issue on artificial intelligence and software engineering).
11. C. Rich, "A Formal Representation for Plans in the Programmer's Apprentice," *Proc. Seventh Int'l Joint Conf. Artificial Intelligence*, Morgan Kaufmann, Los Altos, Calif., 1981, pp. 1044-1052.
12. G. Gruman, "Study Criticizes Defense Dept. Development, Acquisition," *IEEE Software*, Vol. 5, No. 1, Jan. 1988, p. 87.

(*Editor's note:* References 1, 2, 7, 9, 10, and 11 are reprinted in *Readings in Artificial Intelligence and Software Engineering*, C. Rich and R.C. Waters, eds., Morgan Kaufmann, Los Altos, Calif., 1986.)



Charles Rich is a principal research scientist at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology, where he has worked since 1980. He and coauthor Waters are the principal investigators of the Programmer's Apprentice project. Rich's research interests are knowledge representation and the application of artificial intelligence to engineering problem solving, especially in software engineering.

Rich received a bachelor's degree in engineering science from the University of Toronto and master's and doctor's degrees in artificial intelligence from MIT. He is a member of the Assoc. for Computing Machinery and the American Assoc. for Artificial Intelligence as well as the Computer Society.

Richard C. Waters is a principal research scientist at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology, where he has worked since 1978. He and coauthor Rich are the principal investigators of the Programmer's Apprentice project. Waters' other interests include programming languages and engineering problem solving.

Waters received a bachelor's degree magna cum laude in applied mathematics (computer science) from Brown University, a master's degree in computer science from Harvard University, and a doctor's degree in artificial intelligence, with a minor in linguistics, from MIT. He is a member of the Assoc. for Computing Machinery and the American Assoc. for Artificial Intelligence and a senior member of the Computer Society.

Readers may write to the authors at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139.

Was Your Last Software Project Late?

If your last software project was late, you need Costar, a software cost estimation tool that will help you plan and manage your *next* project. Costar is based on the COCOMO model described by Barry Boehm in *Software Engineering Economics.*

COCOMO is used by hundreds of software managers to estimate the cost, staffing levels, and schedule required to complete a project—it's reliable, repeatable, and accurate.

Costar estimates are based on 15 factors that strongly influence the effort required to complete a project, including:
- The Capability and Experience of your Programmers & Analysts
- The Complexity of your project
- The Required Reliability of your project

Costar is a complete implementation of the COCOMO "detailed" model, so it calculates estimates for all phases of your project, from Requirements through Coding, Integration and Maintenance. Costar puts you in control of the estimation and planning process, and provides full traceability for each estimate. User definable cost drivers and a wide variety of reports makes Costar flexible and powerful.

Costar also provides extensive support for "what-if" analyses. You can explore alternative project plans, and compare them side-by-side.

Costar runs on the VAX and IBM PCs.

Softstar Systems
(603) 672-0987
28 Ponemah Road,
Amherst, NH 03031

SOFTST★R

olivetti research

Software Technology Laboratory

If you are interested in working with a small but expanding group of research-oriented people who influence the direction of their lab, then consider joining us at Olivetti Research/STL. Our primary interest is the research and development of technology that will increase the productivity of application developers. Current work includes: a graphical editor for user interfaces, a programming environment for object-oriented languages, support for persistent data and RPC, and a window-based interface to a telephone PBX. In conjunction with the Olivetti research lab in England, we are conducting experiments in video-based electronic mail and video conferencing.

If you are interested and have a PhD in Computer Science (or equivalent experience), send a resume and cover letter to

Dr. Carl Dellar, Director
Olivetti Software Technology Laboratory
5 Palo Alto Sq., Suite 910
Palo Alto, CA 94306