Artificial Intelligence and Software Engineering

David Barstow

Schlumberger-Doll Research
Old Quarry Road
Ridgefield, Connecticut 06877-4108

Abstract

Software Engineering is a knowledge-intensive activity, requiring extensive knowledge of the application domain and of the target software itself. Many Software Engineering costs can be attributed to the ineffectiveness of current techniques for managing this knowledge, and Artificial Intelligence techniques can help alleviate this situation. More than two decades of research have led to many significant theoretical results, but few demonstrations of practical utility. This is due in part to the amount and diversity of knowledge required by Software Engineering activities, and in part to the fact that much of the research has been narrowly focused, missing many issues that are of great practical importance. Important issues that remain to be addressed include the representation and use of domain knowledge and the representation of the design and implementation history of a software system. If solutions to these issues are found, and experiments in practical situations are successful, the implications for the practice of Software Engineering will be profound, and radically different software development paradigms will become possible.

Introduction

Since Software Engineering presumably requires intelligence, it would seem natural to use Artificial Intelligence techniques to build systems to perform or assist in the process. For over two decades, in fact, there has been substantial research with this goal in mind, and there have been some significant results. However, there have been few demonstrations of practical utility. In this paper, I will discuss in some detail the state of the art in the application of AI techniques to SE activities. I will argue that AI techniques are necessary, even though they haven't yet proved successful. I will suggest some reasons for the lack of success, and some directions to pursue in order to achieve more success. Finally, I will discuss the implications that success would have on the practice of Software Engineering.[1]

Artificial Intelligence Techniques

Rather than quibble about what is or is not AI, let me just list four techniques whose history includes contributions from people who thought they were working on AI:

- *Heuristic Search*: Many problems can be formulated as the exploration of a space of alternatives. Heuristics, or informal guidelines, are often used as a way of reducing the amount of the space that must be explored.
- *Formal Inference Systems*: In a sense, most AI systems draw inferences based on formal rules, but there are two specific types that are important for software development systems: theorem-provers and algebraic manipulation systems.
- *Rule-based Systems*: Most so-called expert systems consist of a body of rules and an inference engine that is used to apply the rules to a particular problem.
- *Knowledge Representation Systems*: Knowledge representation systems provide a variety of techniques for explicitly describing knowledge about a domain or a problem. Common techniques include taxonomic hierarchies and part-whole relationships, in addition to rules. A representation system may include mechanisms for drawing inferences from the stored knowledge, for explaining the knowledge and inferences to a human user, and to help in acquiring new knowledge.

The major focus of AI research for more than the last decade has been on techniques for representing and using knowledge, as suggested by the last techniques on the list. This is quite appropriate from the point of view of SE activities, since the major argument for applying AI techniques to SE activities is that SE activities are knowledge-intensive. Let us consider this argument in more detail by looking at specific SE activities to identify the types of knowledge required and the ways the knowledge is used.

Software Engineering Activities

SE activities are usually grouped into two major categories [23]. *Programming-in-the-small* is concerned with the efforts of individual programmers to develop moderate-sized programs (a few thousand lines) within a short period of time (a few months). *Programming-in-the-large* is concerned with the efforts of groups of people to develop large software systems with long intended lifetimes. Both categories require large amounts of diverse knowledge, as illustrated by two examples.

[1] There are, of course, numerous other interactions between AI and SE, including indirect contributions of AI centers to software systems (e.g., time-sharing), the utility of AI techniques as programming paradigms (e.g., rule-based systems), and the need to apply SE standards and techniques to AI systems. However, each of these topics deserves a presentation at least as long as this one, and will therefore not be considered further here. For other discussions of the relationship between AI and SE, see Simon's presentation at the Seventh International Conference on Software Engineering [55], the special issue of *IEEE Transactions on Software Engineering* [44], and several collections of papers [12,16,52].

Reviewed and recommended by:
Kouichi Kishida and Kouji Torii

Programming-in-the-Small The first example is concerned with oil well logging tools which are used to measure various physical properties of an oil well and the surrounding rock formations in order to identify the type of rock and the fluids in it. A logging tool is controlled by a computer on the earth's surface which communicates with the tool through a long cable. In addition to controlling the tool's behavior, the software must record the data from the tool's sensors for later analysis. The software for any given tool is typically a few thousand lines of code and may require a few months to a year to develop.

The example problem is to write the software for a newly developed logging tool. In attacking this problem, a software developer typically goes through the following steps:

1. *Specification:* Develop a description of what the software is supposed to do. This essentially involves studying the tool and the physics upon which its measurements are based. It is usually necessary to communicate extensively with the tool's designer.
2. *Decomposition:* Decompose the specification into relatively independent pieces that are more manageable than the complete specification. This obviously requires knowledge about the specification. In addition, it requires knowledge about program structuring techniques and about some of the characteristics of the computer upon which the software will be executed. It also involves substantial interaction with the tool designer in order to clarify ambiguities, inconsistencies, or missing information in the specification.
3. *Implementation:* Write code for each of the components. This requires knowledge about programming techniques and about the target machine. It also requires considerable interaction with the tool designer.
4. *Testing:* Test whether the code is an accurate implementation of the specification. This requires knowledge of the specification and the tool and will again require interaction with the tool designer.
5. *Optimization:* Ensure that the software satisfies the real-time constraints imposed by the communication system and the tool's physics. This is one of the most time-consuming activities since it involves a substantial amount of measurement and testing. It requires a significant amount of knowledge about the tool and its physics, and also requires substantial interaction with the tool designer.
6. *Validation:* Ensure that the software is what is really desired. Despite the best efforts of the developers, the combined software/tool system usually does not produce the desired results on the first try, largely due to uncertainties about how the tool interacts with the borehole environment. Thus, the specification and the code must be changed, requiring additional testing.
7. *Evolution:* Change the software to reflect improvements in the tool and our understanding of physics. This obviously requires knowledge of the tool and physics. It also requires substantial knowledge about the implementation decisions that went into the initial version of the code. This usually requires considerable interaction with the person who wrote the original version of the software, although that person may not always be available. Because of such problems, the effort devoted to evolution may exceed the effort devoted to the initial development.

Note that all of these activities involved considerable knowledge about the tool and its physics, as well as knowledge about earlier software development activities.

Programming-in-the-Large The second example is concerned with the software that processes tax returns. Specifically, we can imagine what is probably happening at the Internal Revenue Service right now, as it attempts to develop the software for the new tax law that was passed in 1986.

1. *Requirements Analysis:* Try to understand the new tax law, almost 2000 pages of legal documents that no doubt include a large number of ambiguities and inconsistencies. This activity requires a major effort by a large number of people, including many tax experts as well as software developers.
2. *Design:* Develop an overall structure for the system. This activity also involves a large number of people and requires knowledge about software engineering methodologies. It also requires significant interaction with tax experts, both to clarify the law and the requirements document and to help ensure that the design anticipates future changes in the tax law.
3. *Coding:* Write source code for the various components of the system. The activities of the coders can proceed somewhat independently, but they will no doubt need to interact with the designers and possibly with the tax experts.
4. *Integration:* Put the components together into the complete system. This is again a group activity, requiring considerable interaction among the designers and coders.
5. *Testing and Validation:* Test the system to ensure that it accurately reflects the tax law. Given the complexities of the law, this is likely to be an extremely time-consuming activity, requiring interactions among designers, coders, and tax experts.
6. *Maintenance and Evolution:* Change the system to reflect changes in the law. The law is likely to be changed many times before a major new law takes effect. In fact, the first changes are likely to happen before the tax return processing system has been delivered in the first place. As was the case with tool software, this activity will become the most time-consuming over the life-cycle of the system.

Note that knowledge about the tax law played a role in all of the activities and that knowledge about decisions made in early activities was required in later activities. Note also that most of the activities involved a considerable amount of interaction among the various people involved.

Types and Uses of Knowledge Several types of knowledge are obviously required for SE activities. Knowledge of programming techniques is clearly required for programming-in-the-small. Similarly, knowledge of software engineering methodologies is required for programming-in-the-large. Knowledge about the architecture of the target machine is required for both.

Two other types of knowledge are less obvious but perhaps more interesting. Knowledge of the application domain is used extensively in both programming-in-the-small and programming-in-the-large. Knowledge about the target software itself, including the design and implementation decisions and the motivations for them, is also required, especially during maintenance and evolution. Table 1 summarizes the activities in which these last two areas of knowledge play substantial roles.

Activity	Area of Knowledge		
	Application Domain	*Target Software*	
Programming-in-the-small			
Specification	√	•	
Decomposition		•	
Implementation		•	
Testing	√		√
Optimization	√	•	√
Validation	√		√
Evolution	√	•	√
Programming-in-the-large			
Requirements Analalysis	√	•	
Design	√	•	
Coding	√	•	√
Integration			√
Testing and Validation	√		√
Maintenance and Evolution	√		√

√ = activity uses substantial knowledge of area
• = activity creates substantial knowledge for later use

Table 1: Knowledge used in SE activities

In addition to noting the areas of knowledge, we may also characterize the ways that the knowledge is used:

- *Decision-making*: Decision-making is central to many of the activities, especially implementation and design.
- *Inference and analysis*: It is frequently necessary to infer and analyze the properties of the domain or of the target software. For example, requirements analysis is based on inferences about the domain, and testing and optimization require analysis of the properties of the program being written.
- *Communication*: The •'s and √'s in Table 1 suggest important knowledge communication paths, especially knowledge about various decisions and the motivations for them.

The Role of AI Techniques in SE Activities

Why AI Techniques Are Necessary

In considering Table 1, it is important to observe that large amounts of knowledge in two areas are rarely recorded explicitly. Knowledge of the domain is reflected in specifications and requirements documents, but it is rare to record the reasoning behind the specifications and the requirements. Similarly, most implementation and design decisions are left unstated, and those records that are kept (e.g., comments, documentation) usually do not include any discussion of the motivations for the decisions. Thus, software developers usually work in a context of great uncertainty about these two important areas of knowledge. This, in turn, leads to major problems for the developers and therefore substantial software development costs. These uncertainties can only be removed by providing computer support for more effective management of the knowledge. Such computer support must in turn rely on AI techniques. Thus, the basic argument for applying AI techniques for SE activities may be summarized as follows:

- **Software Engineering activities are knowledge-intensive.**
- **Current techniques for managing the relevant knowledge lead to great uncertainty, and therefore large costs.**
- **Removing the uncertainty requires computational support in the form of explicit representation of the relevant knowledge.**
- **Computational support for effective representation and use of knowledge requires Artificial Intelligence techniques.**

To me, this argument alone is sufficient to demonstrate the necessity of AI techniques to support SE activities. However, we can be more specific about the uses of the four AI techniques listed earlier:

- *The heuristic search paradigm can be used to model the design and implementation activities.* One advantage of this technique is that all decision points are explicitly identified by nodes in the space where alternative paths may be taken.
- *Formal inference systems are needed to analyze and infer properties about both the domain and the target software.* Such systems must, of course, be coupled with the knowledge representation systems used to store the knowledge about the domain and the software.
- *Formal inference systems are needed to embody mathematical techniques.* Such techniques are needed primarily during implementation and optimization.
- *Knowledge representation techniques in general, and rule-based systems in particular, are needed to represent and use expertise about SE methodologies and programming techniques.* Such expertise is required for virtually all of the activities.
- *Knowledge representation techniques are needed to store and retrieve knowledge about the application domain.* This is especially necessary during specification, requirements analysis, and evolution.

- *Knowledge representation techniques are needed to store and retrieve knowledge about the history of the software.* This knowledge is created during specification, implementation, requirements analysis, and design. It is especially important to record the motivations for the various decisions made during these activities. The resulting knowledge must be retrieved during most of the other activities.

We can also be more specific about AI techniques that have not yet been developed or characterized as clearly as the four listed earlier:

- *Knowledge acquisition techniques are necessary during specification and requirements analysis.* We cannot assume that every domain will have been represented completely before anyone tries to do requirements analysis or specification.
- *Knowledge explanation techniques are necessary during maintenance and evolution.* So much knowledge must be recorded during design and implementation that simply retrieving it will be inadequate. The retrieval process itself will depend on the needs of the user and the uses to which the knowledge will be put.

Thus, I believe the case for the necessity of AI techniques to support SE activities is quite strong.[2]

Why AI Techniques Are Not Enough

Of course, AI techniques aren't enough to solve the SE problem. One reason is that practical AI systems require much more than AI techniques, as has been clearly demonstrated in work on expert systems. For example, the "AI" part of the Dipmeter Advisor[3] actually accounts for only 30% of the code. The other major parts include signal processing algorithms (13%), user interface (42%), and support environment (15%) [59].

The second reason is that other techniques would be needed to support SE activities, even without AI techniques. The list of necessary techniques includes at least the following:

- *Data base techniques are required to store and retrieve the knowledge created during system development.* In fact, what is probably most important here is a combination of data base and knowledge representation techniques.
- *Communication systems are required to facilitate the coordination of large groups of people in the context of programming-in-the-large.* It is already clear that networking and protocols are needed, especially when much of the work is done on individual workstations.
- *User interface techniques are required to facilitate the interaction between software developers and their development environments.* Sophisticated graphics clearly have a role to play here, but it is likely that other techniques will also be required. A merging of interface and AI techniques is probablay the best strategy (e.g., user modeling, knowledge-based interfaces [61]).

In other words, AI techniques are needed, but so are many other techniques. And they will all be most effective if they are applied to SE activities in a coordinated and coherent way.

Past Research

I would now like to briefly survey the last twenty years of research aimed at applying AI techniques to SE activities. I will not try to provide a neat taxonomy, or comparison along fixed dimensions. Rather, I will try to identify and characterize the main threads of research.

Program Verification

The goal of program verification is to prove mathematically that a program satisfies a formal specification, usually stated in terms of preconditions on input terms and postconditions on input and output terms. The basic strategy is to derive a set of verification conditions from the specifications and the code, and to use a theorem prover to prove that the conditions hold. The current state of the art is that small programs, up to a few thousand lines, can be verified, but that a substantial amount of effort, both human and machine, is required. The major problems relate to the difficulty of developing formal specifications, the number and complexity of the verification conditions, and the difficulty of guiding a theorem-prover when the theorem does not correspond in a natural and intuitive way to the code or the specifications. Despite these difficulties, the effort may be justified if the software is to be run in situations requiring extreme reliability. For more detailed discussions of program verification techniques, see the proceedings of a recent *Verkshop* [67].

Deductive Synthesis

Deductive synthesis is based on a correspondence between writing a program and proving a theorem. A logical formula is derived mechanically from a specification; if a constructive proof of the formula can be found by a theorem-prover, then a program can be derived mechanically from the proof. The foundations of the technique were laid in the late 1960's [28,68] but its roots go back to the Heuristic Compiler [54]. The current state of the art is that small algorithms can be written automatically using the techniques [42]. Perhaps the greatest difficulty is that the space searched by the theorem-prover is not intuitively related to the space of possible programs. This is especially problematic for synthesizing efficient programs, since efficient programs may not correspond to short proofs. Recent work in this area has begun to address the drawbacks by focusing on particular algorithm design techniques [15,43,57].

Transformational Synthesis

The central idea in transformational synthesis is that the programming process can be viewed as a sequence of transformations that change the specification into a program. Specifications are typically in the form of a very high level language involving

[2] This argument assumes that SE activities, or at least something analagous to traditional programs and software systems, will continue to be important. It is imaginable that the future holds something quite different, such as software systems whose internal mechanisms are represented explicitly within the systems themselves [71], or that alternative programming paradigms (e.g., object-oriented programming [27,60]) will radically change the nature of software development. I don't expect such revolutionary changes, but even if they occur, the issues discussed here will still be relevant.

[3] Mark of Schlumberger

mathematical constructs such as sets, mappings, relations, and constraints. A variety of transformations may be employed, including function folding and unfolding, data type refinement, and optimizing transformations. Since the transformations preserve correctness, the transformation process itself guarantees the correctness of the resulting program. Since several transformations may be applicable at each step, the transformations actually define a tree, with different paths leading either to deadends or to different implementations. Thus, the central problem is to explore the space of implementations to find a complete path to an implementation that satisfies some preference criterion, such as efficiency. The experimental roots of this technique go back to PDS [22] and TAMPR [19], and the theoretical foundations were first studied by Burstall and Darlington [21]. Since that early work, several approaches to using transformations have been explored.

One approach, typified by the CIP project [14], has focused on formal characterizations of the transformation and the wide-spectrum language used to describe the program in its intermediate states [13]. The current state of their work is that a prototype implementation of CIP has been used interactively to develop the system itself.

A second approach, typified by the work at ISI [2], has focused on transformations that are intuitively clear to a human user [3] on automating certain aspects of transformation selection (GLITTER [25]), and on describing the particular sequence of transformations used on a specific problem, so that they may be reapplied when the specification is changed (PADDLE [70]).

The third approach, first explored in the synthesis phase of the Ψ project [29,39], has focused on the use of transformations as a representation for detailed knowledge about programming techniques (PECOS [8]) and on the use of analytic and heuristic techniques to find an efficient implementation within the space defined by the transformations (LIBRA [37]). The initial work on Ψ included transformations for refining abstract algorithms in the domain of symbolic programming. More recently, the CHI project [32,58] has built on and extended the techniques.

In addition to these implemented systems, the transformational paradigm has been used in Gedanken experiments to analyze and understand specific algorithms and families of algorithms [10,15,30,53].

All of these approaches to transformational synthesis are still being actively explored. For a more detailed discussion of a variety of transformation systems, see the survey by Partsch and Steinbrüggen [47].

Plan-Based Assistants

The central idea of a plan-based assistant is that many programs can be viewed as the composition of standard program fragments or "plans". This idea goes back at least as far as Winograd's proposal for system "A" [72], and has been the focus of the Programmer's Apprentice project at MIT for the last decade. The original vision was that the PA would participate in the program development process, analyzing the human programmer's code to offer advice [51]. In recent work on a system called KBEmacs, the emphasis has shifted from an analysis to synthesis [69]. In both cases, the system's knowledge of programming is represented in a formalism referred to as the Plan Calculus [50]. Several other projects have built on these foundations. For example, PROUST uses a similar technique for representing and using programming knowledge in the context of a tutor for novice programmers [36].

Natural Language Specifications

The idea of specifying a program by describing it in a natural language like English is attractive for several reasons: it reduces the need for computational sophistication on the part of the specifier; it permits a level of informality; it facilitates communication with other people. An early effort in this direction involved using natural language to define domain terms that could then be translated into a data base formalism [5]. An effort at IBM was aimed at using natural language to specify certain types of simulation programs [33]. Natural language was also one of the specification techniques used in the acquisition phase of the Ψ project [26]. One interesting variation is to use a natural language paraphrase to assist in validating a formal specification [64]. None of these efforts have progressed beyond the experimental stage.

Programming By Example

Another attractive specification technique is to use examples. The basic idea is to use inductive inference techniques to infer a program from examples of input/output pairs. A variation on the theme is to infer the program from sample traces. The central problem, of course, is to infer the "right" program from the infinite set of programs that would compute the outputs from the inputs. Most techniques use some measure of simplicity. Such techniques have been used successfully for some small list-processing programs [56]. As with natural language specifications, none of this research has progressed past the experimental stage.

Domain-Specific Automatic Programming

By restricting the application domain, it is often possible to exploit particular features of the domain [7,9]. The simplest form for such a system is a template-based program generator that does not incorporate AI techniques [35]. AI techniques are needed in more complex situations. For example, ϕ_0 was a domain-specific automatic programming system for programs that interpret oil well log data [11]. It included an algebraic manipulator to transform implicit systems of equations into explicit systems, as well as a graphical specification system that embodied knowledge about the domain. Other domains have also been address with some success, such as business data processing systems [48]. The DRACO project is related, but the focus has been on general techniques for transforming concepts from one domain into another, rather than on exploiting characteristics of any specific domain [45]. In general, domain-specific systems can be useful within their range of application, but the range is often quite narrow.

Other Work

In addition to the research threads just described, several other intersting attempts to apply AI techniques to SE activities should be mentioned:

Katz and Zimmerman-Gal have developed an advisory system to assist human programmers in selecting data structures [40]. This system is currently in use in an educational setting.

The XPLAIN system incorporated techniques for explaining the behavior of a program by examining the expert knowledge from which the program was derived [65]. The XPLAIN experiments were done in the domain of digitalis therapy.

The DESIGNER project used psychological modeling and protocol analysis techniques to characterize the behavior of human algorithm designers [38]. This characterization was the basis of attempts to automate the design of certain algorithms [63].

RML is a formalism for expressing software requirements using knowledge representation techniques [18]. It has been used to describe the requirements for several small information systems.

Lessons From Past Research

What Has Been Learned

In summarizing the research just described, we can point to several important results:

- *Programs have a mathematical structure.* Work on verification and deductive synthesis, combined with other work on the semantics of programming languages, has provided strong theoretical and formal foundations.
- *Important computational properties of a program can be represented in machine-usable form.* The Plan Calculus was developed specifically to separate data and control flow, as well as to represent some of teleological properties of a program.
- *Knowledge about algorithm design techniques can be represented in a machine-usable form.* Work on deductive synthesis, especially Smith's recent work, demonstrates this in a formal way. Kant and Steier's recent work shows promise in a less formal approach.
- *Knowledge about programming techniques can be represented in a machine-usable form.* Work on PECOS and the Programmer's Apprentice, as well as the work by Katz and Zimmerman, demonstrate this by providing catalogues of specific programming techniques.
- *Some of the apparently "creative" aspects of programming can be understood in terms of the composition of simpler techniques.* This has been demonstrated in a semi-automatic situation by PECOS and in hand-developed derivations of sophisticated algorithms.
- *The programming process for moderate-size programs can be modeled as a process of applying transformations to a high-level specification.* This was demonstrated by Ψ's synthesis phase, by some of the work at ISI, and by recent work on KPEmacs.
- *The transformational process can be structured in a more useful way than simply a linear sequence of transformation steps.* This was demonstrated by PADDLE and GLITTER at ISI.
- *Writing small programs involves many decisions.* For example, a hundred-line program written by PECOS involved about a thousand rule applications, of which about 30 represented significant implementation decisions.
- *Efficiency considerations can be used to guide search for an efficient program.* LIBRA demonstrated this in the context of algorithm refinement via transformations. Katz and Zimmerman demonstrated this in the context of their data structure advisor.

What Has Not Been Demonstrated

Unfortunately, during two decades of research, there have been virtually no demonstrations of the utility of AI techniques in practical SE situations. With the exception of a few research centers, there are no AI-based software development systems in routine use. In fact, with the possible exception of a few programs verified with great effort, I know of no cases in which such systems have been used to develop software that is in routine use in a practical situation. Why is this the case? In part, of course, it is simply that it's a very hard problem. But we can be more precise, both about the nature of the problem and the way it has been attacked.

It may be helpful to compare the situation with that of other intellectual activities to which AI techniques have been applied with more success, such as medicine. If we compare books, the traditional repositories of knowledge, it would seem that there is more medical knowledge than SE knowledge. However, there seem to me to be four key differences:

- *Suitable abstractions for describing complex software systems have not been developed.* Medicine is concerned primarily with the human body, an artifact that is certainly more complex than any software system we can build today. However, over the centuries doctors have developed frameworks and concepts that allow many of the complexities to be ignored. In a quarter of a century, we have not yet succeeded in identifying similarly useful abstractions for software systems. Our situation is analagous to that of doctors who speak only at the chemical or molecular levels.
- *There is great variation among software systems.* Medicine is concerned with essentially a single type of system, with only two basic variations. SE is concerned with an enormous variety of systems, each of which has important unique characteristics.
- *SE activities require considerable knowledge of the application domain.* Every software system is intended to be used in some real-world situation. In order for the system to be effective, it must be developed with that context in mind. The analagous situation for a doctor would be the need to consider all of the details of the environment in which the patient lives. While that may be necessary to a degree for human doctors, medical expert systems have focused on tasks that need relatively little knowledge of the patient's environment.
- *SE activities are concerned with the design and implementation history of the software system.* The •'s in Table I indicate the activities in which a large number of decisions are made. The √'s indicate activities in which knowledge of those decisions is needed. As I indicated earlier, these later activities require access not only to the decisions but also to the motivations for making them. To carry the medical analogy to an extreme, it is like requiring a doctor to understand why the human body evolved as it did.

Thus, one of the main reasons that there have been few practical demonstrations is the amount and variety of knowledge required in practical SE situations. Another reason is that much of the research has been focused on relatively minor problems:

- *Programming is more than algorithm design.* A large fraction of the research described earlier was focused on the intellectual problems involved in designing algorithms, with some attention paid to data structure selection. Yet most practical software involves algorithms no more difficult than sorting, and when algorithms are needed, most programs just invoke standard procedures. The more important parts of programming-in-the-small involve understanding what the software should do in the first place, understanding the interactions among different parts of the program, and coping with a very large number of details, such as input/output and exceptional conditions.
- *Software Engineering is more than programming.* This point was made earlier in the discussion of programming-in-the-large. Of the effort in building large systems, very little involves programming. A much greater part involves communication among users, developers, and maintainers. However, very little AI research has been aimed at supporting such communication.

The problem can be seen clearly by considering the major time-consuming activities in the example situations described earlier. Table 2 shows the major time-consuming activities and those which involve significant algorithms and data structures.[4]

Activity		
Logging Software		
Learning about tool and physics	•	
Learning about target machine		
Writing signal processing algorithms		√
Writing code for computations and input/output		√
Testing and validating functionality	•	
Measuring performance	•	
Studying previously written code	•	
Tax Software		
Learning about tax law	•	
Designing overall structure		
Designing data base and communications system		√
Writing code for calculations and input/output		√
Testing components and system	•	
Studying previously written code	•	
Communicating with colleagues	•	

• = major time-consuming activity
√ = significant use of algorithms and data structures

Table 2: Activities involved in example situations

Research Directions for Practical Results

Major Issues

Based on the work done to date and on the analysis of SE activities given earlier, it is possible to identify several major conceptual issues that must be addressed if practical results are to be achieved, and in fact, some work has been done on many of them.

[4]Table 2 is not based on quantitative studies. Rather it represents qualitative judgements on my part. Detailed studies might show different bottlenecks, but it seems clear that algorithm design and data structure selection are not central to the major time-consuming activities.

Programming-in-the-Small I see five major research issues that must be addressed in order to develop AI-based systems to support programming-in-the-small:

- *Specification languages:* In specifying a program, a variety of aspects must be described. Most work on specification has focused on specifying the values of output terms. In addition, much practical software includes behavioral and temporal aspects. Further, permissable assumptions must be described, as well as the actions to be taken when assumptions are violated. Finally, the interfaces between the program and its environment must be described, including other programs, external devices and databases, and human users. The most common specification techniques are pre- and postconditions [6,42] and very high level languages [4,24,58]. It seems quite likely that a variety of other approaches will also be required, including natural languages, and graphical and iconic languages, in addition to formal textual languages. Finally, domain-specific languages are likely to grow in importance (e.g. [11]). The entire issue is complicated by the fact that the specification process must be interactive and iterative [66].

- *Codification of programming knowledge:* Much work is needed in order to codify knowledge about specific algorithms and data structures. Such work has been done in a few areas, notably symbolic computation [8,50], sorting [30], and data structures [8,40]. The key here is to pick areas that are reasonably well understood (e.g., for which textbooks exist) and try to represent the knowledge in some machine-usable form. Specific areas that I find attractive are graph algorithms, numeric algorithms, and communication systems. A factor that complicates this work is parallelism: it would be extremely valuable if the knowledge were codified in a way that made explicit the potential uses of parallelism. Unfortunately, this goal also makes the work much more difficult. It would also be valuable to codify knowledge about general programming strategies, such as divide-and-conquer [57]. Other interesting strategies to codify include dynamic programming, linear programming, and search techniques.

- *Techniques for controlling search:* The transformational paradigm seems to be the most promising approach to program synthesis systems. In order for this paradigm to be automated, effective techniques for controlling the search must be developed. Clearly, performance analysis is one such technique, but its effectiveness depends on the ability to do the analysis relatively early in the transformational process. In addition to quantitative analysis, heuristic techniques will presumably also be needed. For both techniques, the role of domain knowledge is not well understood. Kant's work is a good starting point, but significantly more is required [37]

- *Modeling the interactions among program components:* The fact that a program consists of a number of interacting components is a major source of complexity during decomposition, implementation, optimization, and maintenance. It will clearly be necessary to represent the interactions explicitly somehow. One complication is that the components of small programs are tightly intertwined. The plan representation of the Programmer's Apprentice includes some techniques for representing such interactions [50]. SE research has also identified important concepts, such as information hiding [46] and abstract data types [41].

- *Representing the implementation history of a program:* As indicated earlier, a large fraction of the implicit knowledge about a program consists of the decisions that were made during decomposition and implementation, as well as the motivations for them. The transformational paradigm provides a framework for representing those decisions explictly: a decision is made every time that alternative implementation paths are considered, and the facts about the decision can be recorded. The problem is that an enormous number of decisions are made. Thus, we need a more effective way of representing and accessing the information than simply a linear sequence of transformations. Recent work at ISI [25,70] represents a good start in this direction. Interestingly enough, it may not actually be necessary to solve this problem: if it becomes possible to automate programming-in-the-small completely, then there may be no need to record or recall the decision history for the sake of human users. However, complete automation seems unlikely in the near- or mid-term future, and in any case, automation will probably require that similar records be kept for use by the machine.

Programming-in-the-Large In my opinion, four major issues must be addressed with respect to programming-in-the-large[5]:

- *Representation of domain knowledge:* As suggested earlier, knowledge of the domain is extremely important. In fact, requirements analysis is largely an exercise in understanding the domain. To date, however, we have very little experience with explicit representations of that knowledge in the context of software development. On the other hand, there is no reason that domain knowledge for software development should differ substantially from domain knowledge represented for other uses. Thus, we should be able to benefit from most of the work now underway in knowledge acquisition. The one major difference may be in size: the amount of domain knowledge needed during requirements analysis for large software systems is probably much larger than that required for AI systems now under development.
- *Modeling the interfaces among system components:* The issue here is similar to that in programming-in-the-small, except that the components are much larger. However, the major considerations are somewhat different, since characteristics of the domain often play a crucial role. For example, it is helpful to isolate those components related to volatile aspects of the domain, in order to simplify future maintenance and evolution. It is also necessary, of course, to specify the computations to be performed by the components, which will require addressing the specification languages issue discussed in the previous section.
- *Modeling the design history of a system:* This issue is also similar to that of programming-in-the-small, but significantly more complicated. It is unlikely that the transformational paradigm will extend to the design aspects of large systems, so some other paradigm will be needed. An important property of such a paradigm will be that decision points are explicit. It will also be necessary to retrieve relevant facts easily. Given the enormous number of potentially relevant facts, this will be a difficult problem. The best starting point is probably work on designing other artifacts, such as digital circuits and mechanical systems.
- *Modeling collaborative work:* A very large fraction of the time spent in programming-in-the-large involves communication among colleagues, either during development or during evolution. Currently, the primary forms of computer support for such collaboration involve text documents and data bases about the status of various subprojects. It is clear that substantially more support could be provided. The explicit representation of design and implementation decisions, as discussed above, is one form of support. In addition, it should be possible support the various ways that people break tasks into subtasks and communicate about their progress and results. Some early work is already underway on the development of models of such collaborative work [62,1].

Given the complexities of these issues, the best way to address them is probably through case studies with the hope of developing a better understanding of their important aspects. Although not intended as an experiment in applying AI techniques to SE activities, the analysis of the A7 software suggests the type of case study that will be required [34].

Experimental Systems in Practical Situations

In many ways, the situation with respect to AI applied to SE is remiscent of AI research in the years just prior to the development of DENDRAL, MYCIN, and MACSYMA. The focus has been on general techniques rather than on specific applications, and there have been very few attempts to apply AI techniques in practical SE situations. The field of AI reaped tremendous benefits from the paradigm shift that occurred in the late 1960's, in terms of intellectual excitement and public recognition, as well as a better understanding of the computational mechanisms underlying intelligent behavior. We would reap similar benefits. In my opinion, the greatest would be to ensure that we pursue directions that are important. For example, the overemphasis on algorithm design was probably a direct result of pursuing the research outside the context of practical situations. Thus, I believe that the most important research directions to pursue involve building experimental AI systems in practical SE situations. A variety of such experiments are possible and reasonable given what we already know:

- *Domain-specific automatic programming systems:* As noted earlier, some work has already been done in this area. The key problem is to select a domain that is broad enough to be useful, yet restricted enough to be solvable. It is, of course, also helpful if the domain is rich enough to be interesting and potentially generalizable to a broader domain. Several such projects are currently underway, including the ΦNIX project at Schlumberger-Doll Research [6,7] and the ISFI project at Mitre [20].
- *Very high level language compiler:* The transformational paradigm is a reasonable approach to building a compiler for a very high level language. The hope would be to build a general purpose system that is broadly applicable. The penalty for such generality is that search control may not be automatic, relying on user interaction or pragmas. The REFINE™ system, now being marketed commercially, is an example of such a project [49].

[5] The KBSA proposal [31] incorporates many of these issues, and several research projects based on the proposal are now underway.

- *Program library manager:* One of the major problems in software reuse is the difficulty of finding the appropriate program(s) in a library that may be quite large. Simple keyword searching is unlikely to be adequate. With the use of AI techniques for semantic pattern matching, it may be possible to build a relatively successful library manager.
- *Data structure selection advisor:* As mentioned earlier, Katz and Zimmerman-Gal have developed a reasonably sophisticated data structure advisor [40]. An attractive experiment would be to include such a system in the standard programming environment used by programmers in some practical situation. We would learn both about the relative importance of data structure selection in programming and about the issues involved in embedding AI-based systems in traditional programming environments.
- *Maintenance advisor for specific systems:* It is well known that maintenance is the dominant cost in the full life-cycle of a large software system. Unfortunately, it is not feasible to build a general-purpose maintenance advisor, since it would require too much detailed knowledge of the system's design and implementation history. However, it may be possible to build such an advisor system for some specific system, essentially by using "Expert Systems" techniques on the expert maintainers of that particular system. If the system's lifetime is long enough, such an advisory system might reduce maintenance costs substantially.

Given enough experiments along lines such as these, it should be possible to identify the remaining problems with greater certainty and to characterize more general techniques and solutions.

Implications for Software Engineering

Direct Impact on Software Costs

We can do some "back of the envelope" calculations to try to estimate the direct effect that success in some of these research and experimental directions would have on the practice of Software Engineering.

If an automatic programming system were available, we would expect it to drastically reduce the effort required for decomposition, implementation, optimization, and testing, and to lessen the effort required for validation. On the other hand, more effort will probably be required during specification. Thus, the effect on the activities of programming-in-the-small might be something like that shown in the top portion of Table 3, which shows a typical distribution of effort using current methodologies (based loosely on Boehm's COCOMO model [17]) and the effects of an automatic programming system. Thus, we might expect to gain a factor of about 3 overall for programming-in-the-small.

The effect on programming-in-the-large would be less, of course, but still substantial. In particular, we may expect the same factor of 3 for those aspects of programming-in-the-large that are essentially programming-in-the-small; we may also expect a similar gain for that fraction of maintenance and evolution that deals with bugs within small components (perhaps one-fourth). Thus, we may expect a factor of about 1.3 overall, as shown in the bottom portion of Table 3. Table 3 also shows the effects that might be expected from a mechanism for storing and retrieving the design history of a software system. This would have its greatest effect in the remaining fraction of maintenance and evolution, but it would also reduce the effort required during design, since it would simplify communication among the human designers. Thus, an automatic programming system combined with a record of design decions might gain a factor of 2.5 for programming-in-the-large.

Indirect Impact on Software Development Paradigms

Detailed predictions about indirect effects are, of course, difficult to make, but it seems likely that the indirect effects would be substantially greater than the direct effects just described. Current methodologies are based on assumptions about the relative costs of different activities. If the cost of programming-the-small were reduced to an insignificant amount, and if the costs of long-term evolution were reduced substantially, then entirely new SE paradigms would come into being, perhaps even to the point where the activities listed earlier are no longer identifiable parts of the paradigm. One such paradigm might involve the use of

Activity	Current	Auto. Prog.		Des. Hist.		Combined
	Effort	*Factor*	*Effort*	*Factor*	*Effort*	*Effort*
Programming-in-the-small						
Specification	.10	2.0	.20			
Decomposition	.20	0.1	.02			
Implementation	.20	0.1	.02			
Optimization	.15	0.1	.01			
Testing	.25	0.0	.00			
Validation	.10	1.0	.10			
Total Effort	1.00		.35			
Programming-in-the-large						
Requirements Analysis	.05	1.0	.05	1.0	.05	.05
Design	.10	1.0	.10	0.5	.05	.05
Programming-in-the-Small	.15	0.3	.05	1.0	.15	.05
Integration	.10	0.5	.05	1.0	.10	.05
Maintenance and Evolution	.60	0.8	.50	0.5	.30	.20
Total Effort	1.00		.75		.65	.40

Table 3: Approximate effects of AI based tools on SE activities

an automatic programming system to create a large number of very rapid prototypes, in order to significantly reduce the uncertainties about the desired final product. Another such paradigm might involve the partial reuse of design histories, in order to build large systems that are similar but not identical to previously built systems.

Because of such indirect effects, it is difficult to say more than that the effect of successfully applying AI techniques to support SE activities would be profound.

Prospects for Practical Results

What may we say about the prospects for success? How likely is it that there will be practical results? When might they appear?

As suggested earlier, the current situation is similar to that of rule-based systems in 1970. If the analogy holds, then we may guess that several successful experimental systems will be built within the next few years, a few systems will be in routine use in about a decade, and there will be wide-spread use of AI-based systems to support SE activities by the year 2001. In fact, for programming-in-the-small, I think there are reasons to be more optimistic. First, AI techniques themselves are considerably more sophisticated than they were in 1970. Second, we may learn from the commercialization of expert systems: in particular, we now know that practical experimentation is important and that a routinely-used system will have many components that do not involve AI techniques. Third, there have been considerable advances in the technology required for the other components, including data bases, communications systems, and user interfaces. Thus, I would shorten the time scale somewhat, and expect to see significant use of AI-based systems for programming-in-the-small by the mid 1990's. A likely form for such systems will be domain-specific automatic programming systems based on the transformational paradigm, such as ΦNIX.

On the other hand, practical uses of AI techniques in support of programming-in-the-large are probably considerably further off. Of the major research issues mentioned earlier, representing domain knowledge seems most likely to be solved in the near future: within a decade or so, I would hope to see a few practical systems for representing the results of requirements analysis. Unfortunately, the problem of representing the design history of a system seems extremely hard, and I don't expect to see practical AI systems to support the other activities of programming-in-the-large until the next century. As I suggested earlier, the one exception to this pessimistic view might be carefully targeted advisory systems for specific activities and specific projects.

Conclusions

In conclusion, let me try to summarize the main points that I have tried to make:

- *Software Engineering is a knowledge-intensive activity.* Of special importance are knowledge of the application domain and knowledge about the design and implementation history of the target software itself.
- *Artificial Intelligence techniques are necessary in order to build effective systems to support SE activities.* In particular, heuristic search, formal inference systems, rule-based systems, and knowledge representation systems all have roles to play.
- *Despite two decades of research, there have been few practical demonstrations of the utility of AI techniques to support SE activities.* This is due both to the amount and diversity of knowledge that is needed and to the fact that much of the research has been focused on algorithm design, which is relatively unimportant in practical SE situations.
- *For programming-in-the-small, we are ready for substantial experiments in practical situations.* In fact, it is quite important that we do so. To me, the most attractive approach is to develop domain-specific automatic programming systems based on the transformational paradigm, such as the ΦNIX project.
- *For programming-in-the-large, there are several major research issues that must be addressed before there will be practical demonstrations.* These include techniques for representing domain knowledge during requirements analysis, techniques for modeling the interactions among system components, and, most importantly, techniques for modeling the design and implementation history of a software system.
- *If these experimental and research issues are addressed successfully, the impact of AI techniques on the practice of SE will be profound.* Most current SE methodologies are based on assumptions about the relative effort and cost of different activities—AI techniques will permit those assumptions to be changed substantially.

Acknowledgements

The ideas expressed here have grown out of interactions with many people. I especially appreciate my colleagues at SDR and the other Schlumberger centers, including Paul Barth, Paul Dietz, Rick Dinitz, Sol Greenspan, Scott Guthery, Elaine Kant, Dennis O'Neill, Reid Smith, Stan Vestal, and Peter Will. In addition, the paper has benefited from critiquing by Bob Balzer, Les Belady, Ted Biggerstaff, Barry Boehm, and Cordell Green.

References

[1] Association for Computing Machinery. *Proceedings of the Conference on Computer-Supported Cooperative Work*, Austin, Texas, December 1986.

[2] R. Balzer. A 15 year perspective on automatic programming. *IEEE Transactions on Software Engineering*, 11(11):1257–1268, November 1985.

[3] R. Balzer. Transformational programming: an example. *IEEE Transactions on Software Engineering*, 7(1):3–14, January 1981.

[4] R. Balzer, D. Cohen, M. Feather, N. Goldman, W. Swartout, and D. Wile. Operational specification as the basis for specification validation. In Ferrari, Bolognani, and Goguen, editors, *Theory and Practice of Software Technology*, North Holland, Amsterdam, 1983.

[5] R. Balzer, N. Goldman, and D. Wile. Informality in program specifications. *IEEE Transactions on Software Engineering*, 4(1):94–103, February 1978. reprinted in [52].

[6] D. Barstow. Automatic programming for streams. In *Ninth International Joint Conference on Artificial Intelligence*, pages 232–237, Los Angeles, August 1985.

[7] D. Barstow. Domain-specific automatic programming. *IEEE Transactions on Software Engineering*, 11(11):1321–1336, November 1985.

[8] D. Barstow. An experiment in knowledge-based automatic programming. *Artificial Intelligence Journal*, 12(2):73–119, August 1979. reprinted in [52].

[9] D. Barstow. A perspective on automatic programming. *A I Magazine*, 5(1):5–27, Spring 1984.

[10] D. Barstow. The roles of knowledge and deduction in algorithm design. In J. Hayes, D. Michie, and Y.-H. Pao, editors, *Machine Intelligence 10*, Ellis Horwood and Wiley, New York, 1982. reprinted in [16].

[11] D. Barstow, R. Duffey, S. Smoliar, and S. Vestal. An automatic programming system to support an experimental science. In *Sixth International Conference on Software Engineering*, pages 360–366, Tokyo, Japan, September 1982.

[12] D. Barstow, H. Shrobe, and E. Sandewall, editors. *Interactive Programming Environments.* McGraw-Hill, New York, 1984.

[13] F. Bauer, , M. Broy, W. Dosch, F. Geiselbrechtinger, W. Hesse, R. Gnatz, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Samelson, M. Wirsing, and H. Wössner. *Algorithmic Language and Program Development.* Springer-Verlag, New York, 1982.

[14] F. Bauer and H. Wössner, editors. *Algorithmic Language and Program Development.* Springer-Verlag, New York, 1982.

[15] W. Bibel. Syntex-directed, semantics-supported program synthesis. *Artificial Intelligence Journal*, 14(3):243–261, November 1981.

[16] A. Biermann, G. Guiho, and Y. Kodratoff, editors. *Automatic Program Construction Techniques.* Macmillan, New York, 1984.

[17] B. Boehm. *Software Engineering Economics.* Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[18] A. Borgida, S. Greenspan, and J. Mylopoulos. Knowledge representation as the basis for requirements specifications. *IEEE Computer*, 18(4):82–91, April 1985. reprinted in [52].

[19] J. Boyle. Software adaptability and program transformation. In H. Freeman and P. Lewis, editors, *Software Engineering*, pages 75–93, Academic Press, New York, 1980.

[20] R. Brown. *Automation of programming: the ISFI experiments.* Technical Report M85-21, MITRE, Bedford, Massachusetts, 1985.

[21] R. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[22] T. Cheatham, J. Townley, and G. Holloway. A system for program refinement. In *Fourth International Conference on Software Engineering*, pages 53–62, Munich, Germany, September 1979. reprinted in [12].

[23] F. DeRemer and H. Kron. Programming in the large versus programming in the small. *IEEE Transactions on Software Engineering*, 1, 1975.

[24] R. Dewar, A. Grand, S.-C. Liu, and J. Schwartz. Programming by refinement as exemplified by the SETL representation subsystem. *ACM Transactions on Programming Languages and Systems*, 1(1):27–49, July 1979.

[25] S. Fickas. Automating the transformational development of software. *IEEE Transactions on Software Engineering*, 11(11):1268–1277, November 1985.

[26] J. Ginsparg. *Natural language processing in an automatic programming domain.* PhD thesis, Stanford University, 1978. Stanford Report AIM-316.

[27] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, MA., 1983.

[28] C. Green. Application of theorem proving to problem solving. In *First International Joint Conference on Artificial Intelligence*, pages 219–239, Washington, D.C., May 1969.

[29] C. Green. The design of the Ψ program synthesis system. In *Second International Conference on Software Engineering*, pages 4–18, San Francisco, October 1976.

[30] C. Green and D. Barstow. On program synthesis knowledge. *Artificial Intelligence Journal*, 10(2):241–279, August 1978. reprinted in [52].

[31] C. Green, D. Luckham, R. Balzer, T. Cheatham, and C. Rich. *Report on a knowledge-based software assistant.* Technical Report, Kestrel Institute, June 1983. reprinted in [52].

[32] C. Green, J. Philips, J. Westfold, T. Pressburger, B. Kedzierski, S. Angerbranndt, B. Mont-Reynaud, and S. Tappel. *Research on knowledge-based programming and algorithm design.* Technical Report KES.U.81.2, Kestrel Institute, Palo Alto, California, 1982.

[33] G. Heidorn. Automatic programming through natural language dialogue: a survey. *IBM Journal of Research and Development*, 20(4):302–313, July 1976. reprinted in [52].

[34] K. Heninger. Specifying software requirements for complex systems: new techniques and their application. *IEEE Transactions on Software Engineering*, 6(1):2–12, January 1980.

[35] E. Horowitz, A. Kemper, and B. Narasimhan. A survey of application generators. *IEEE Software*, 2(1):40–54, January 1985.

[36] W. Johnson and E. Soloway. PROUST: knowledge-based program understanding. *IEEE Transactions on Software Engineering*, 11(3):267–275, March 1985. reprinted in [52].

[37] E. Kant. On the efficient synthesis of efficient programs. *Artificial Intelligence Journal*, 20(3):253–306, May 1983. reprinted in [52].

[38] E. Kant. Understanding and automating algorithm design. *IEEE Transactions on Software Engineering*, 11(11):1361–1374, November 1985.

[39] E. Kant and D. Barstow. The refinement paradigm: the interaction of coding and efficiency knowledge in program synthesis. *IEEE Transactions on Software Engineering*, 7(5):458–471, September 1981. reprinted in [12].

[40] S. Katz and R. Zimmerman. An advisory system for developing data representations. In *Seventh International Joint Conference on Artificial Intelligence*, pages 1030–1036, Vancouver, British Columbia, Canada, August 1981.

[41] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development.* McGraw-Hill, New York, 1985.

[42] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, January 1980. reprinted in [52].

[43] Z. Manna and R. Waldinger. The origin of the binary-search paradigm. In *Ninth International Joint Conference on Artificial Intelligence*, pages 222–224, Los Angeles, August 1985.

[44] J. Mostow, editor. Special issue on artificial intelligence and software engineering. *IEEE Transactions on Software Engineering*, 11(11), November 1985.

[45] J. Neighbors. The Draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, 10(5):564–574, May 1984.

[46] D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(3), March 1972.

[47] H. Partsch and R. Steinbrüggen. Program transformation systems. *Computing Surveys*, 15(3):199–236, September 1983.

[48] N. Prywes, A. Pnuelli, and S. Shastry. Use of a nonprocedural specification language and associated program generator in software development. *ACM Transactions on Programming Languages and Systems*, 1(2):196–217, October 1979.

[49] *REFINE Users Guide.* Reasoning Systems, Inc., Palo Alto, California, 1985.

[50] C. Rich. A formal representation for plans in the Programmer's Apprentice. In *Seventh International Joint Conference on Artificial Intelligence*, pages 1044–1052, Vancouver, British Columbia, Canada, August 1981.

[51] C. Rich and H. Shrobe. Initial report on a Lisp programmer's apprentice. *IEEE Transactions on Software Engineering*, 4(6):456–467, November 1978. reprinted in [12].

[52] C. Rich and R. Waters, editors. *Artificial Intelligence and Software Engineering.* Morgan Kaufmann, Los Altos, California, 1986.

[53] W. Scherlis and D. Scott. First steps towards inferential programming. In *Proceedings, IFIP 83 Congress*, pages 199–212, Paris, August 1983.

[54] H. Simon. Experiments with a heuristic compiler. *Journal of the ACM*, 10(4), October 1963.

[55] H. Simon. Whether software engineering needs to be artificially intelligent. *IEEE Transactions on Software Engineering*, 12(7):726–732, July 1986.

[56] D. Smith. The synthesis of Lisp programs from examples: a survey. In A. Biermann, G. Guiho, and Y. Kodratoff, editors, *Automatic Program Construction Techniques*, pages 307–324, Macmillan, New York, 1984.

[57] D. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence Journal*, 27(1):43–96, September 1985. reprinted in [52].

[58] D. Smith, G. Kotik, and S. Westfold. Research on knowledge-based software environments at Kestrel Institute. *IEEE Transactions on Software Engineering*, 11(11):1278–1295, November 1985.

[59] R. Smith. On the development of commercial expert systems. *A I Magazine*, 5(3):61–73, Fall 1984.

[60] R. Smith. Strobe: support for structured object knowledge representation. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pages 855–858, August 1983.

[61] R. Smith, R. Dinitz, and P. Barth. Impulse-86: a substrate for object oriented interface design. In *ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, pages 167–176, Portland, Oregon, September 1986.

[62] M. Stefik, G. Foster, D. Bobrow, K. Kahn, S. Lanning, and L. Suchman. Beyond the chalkboard: computer support for collaboration and problem solving in meetings. *Communications of the ACM*, 30(1):32–47, January 1987.

[63] D. Steier and E. Kant. The roles of execution and analysis in algorithm design. *IEEE Transactions on Software Engineering*, 11(11):1374–1386, November 1985.

[64] W. Swartout. GIST English generator. In *Second National Conference on Artificial Intelligence*, pages 404–409, Pittsburgh, Pennsylvania, August 1982.

[65] W. Swartout. Xplain: a system for creating and explaining expert consulting systems. *Artificial Intelligence Journal*, 21(3):285–325, September 1983.

[66] W. Swartout and R. Balzer. On the inevitable intertwining of specification and implementation. *Communications of the ACM*, 25(7):438–440, July 1982.

[67] Verkshop III—a formal verification workshop. *ACM Software Engineering Notes*, 10(4), August 1985.

[68] R. Waldinger. PROW: a step toward automatic program writing. In *First International Joint Conference on Artificial Intelligence*, pages 241–252, Washington, D.C., May 1969.

[69] R. Waters. The Programmer's Apprentice: a session with KBEmacs. *IEEE Transactions on Software Engineering*, 11(11):1296–1320, November 1985. reprinted in [52].

[70] D. Wile. Program developments: formal explanations of implementations. *Communications of the ACM*, 26(11):902–911, November 1983. reprinted in [52].

[71] T. Winograd. Beyond programming languages. *Communications of the ACM*, 22(7):391–401, July 1979. reprinted in [12].

[72] T. Winograd. Breaking the complexity barrier (again). In *Proceedings of the ACM SIGPLAN/SIGIR Interface Meeging on Programming Languages-Information Retrieval*, pages 13–30, Gaithersburg, Maryland, November 1973. reprinted in [12].