

Software Engineering in the Twenty-First Century

Michael R. Lowry

There is substantial evidence that AI technology can meet the requirements of the large potential market that will exist for knowledge-based software engineering at the turn of the century. In this article, which forms the conclusion to the AAAI Press book *Automating Software Design*, edited by Michael Lowry and Robert McCartney, Michael Lowry discusses the future of software engineering, and how knowledge-based software engineering (KBSE) progress will lead to system development environments. Specifically, Lowry examines how KBSE techniques promote additive programming methods and how they can be developed and introduced in an evolutionary way.

By the year 2000, there will be a large potential market and a fertile environment for knowledge-based software engineering (KBSE). In the coming decade, hardware improvements will stimulate demand for large and sophisticated application software, and standardization of software interfaces and operating systems will intensify competition among software developers. In this environment, developers who can rapidly create robust and error-free software will flourish. Developers who deliver buggy software years behind schedule, as is typical today, will perish. To meet this challenge, software developers will seek tools and methods to automate software design.

Computer-aided software engineering (CASE) is undergoing tremendous commercial growth. However, the current generation of CASE tools is limited by shallow representations and shallow reasoning methods. CASE tools will either evolve into, or be replaced by, tools with deeper representations and more sophisticated reasoning methods. The enabling technology will come from AI, formal methods, programming language theory, and other areas of computer science. This technology will enable much of the knowledge now lost in the software development process to be captured in machine-

encoded form and automated. KBSE will revolutionize software design, just as computer-aided design has revolutionized hardware design, and desktop publishing has revolutionized publication design.

This article draws on the chapters in *Automating Software Design* and other sources to present one vision of the future evolution of KBSE. After an executive summary and a brief history of software engineering, the role of AI technology is examined in mainstream software engineering today. Currently, AI programming environments facilitate rapid prototyping but do not produce efficient, production-quality code. KBSE technology combines the advantages of rapid prototyping and efficient code in a new programming paradigm: transformational programming, described in the subsequent part of the conclusion. In transformational programming, prototyping, validation, and modifications are done at the specification level; automatic program synthesis then translates specifications into efficient code. The following part compares the trade-offs in various approaches to program synthesis. Then, several near-term commercial applications of KBSE technology are predicted for the next decade. To scale up from these near-term applications to revolutionizing the entire software life

cycle in the next century, the computational requirements of KBSE technology need to be addressed. An examination of current benchmarks reveals that hardware performance by the year 2000 is not likely to be a limiting factor but that fundamental issues such as search control require further research. Finally, the future of KBSE in the next century is presented from the viewpoint of different people in the software life cycle—from end users to the knowledge engineers who encode domain knowledge and design knowledge in software architectures.

Executive Summary

Currently, KBSE is at the same stage of development as early compilers or expert systems. Commercially, a few dozen handcrafted systems are in real use as industrial pilot projects. The first and third sections of this book describe pilot systems for software maintenance and special-purpose program synthesis. In research laboratories, many prototype KBSE systems have been developed that have advanced the science of formalizing and automating software design knowledge. Program synthesis research has matured over the last two decades to the point that sophisticated algorithms can be synthesized with only limited human guidance. Research in intelligent assistance for requirement and specification engineering is less mature but already shows considerable promise.

The Next Decade

Within the next decade, significant commercial use of KBSE technology could occur in software maintenance and special-purpose program synthesis. The influence will be evolutionary and compatible with current software development methods. Some research systems for intelligent assistance in requirement and specification engineering, such as ARIES and ROSE-2 incorporate many of the current CASE representations. Thus, as they mature, they could be integrated into the next generation of CASE tools. On the research front, I expect continued progress in representing and reasoning with domain and design knowledge. This research will pay off in more sophisticated tools for requirement and specification engineering as well as better program synthesis systems. Within the decade, the break-even point will be reached in general-purpose program synthesis systems,

This article draws on the chapters in *Automating Software Design* and other sources to present one vision of the future evolution of KBSE. After an executive summary and a brief history of software engineering, the role of AI technology is examined in mainstream software engineering today. Currently, AI programming environments facilitate rapid prototyping but do not produce efficient, production-quality code. KBSE technology combines the advantages of rapid prototyping and efficient code in a new programming paradigm: transformational programming, described in the subsequent part of the conclusion. In transformational programming, prototyping, validation, and modifications are done at the specification level; automatic program synthesis then translates specifications into efficient code. The following part compares the trade-offs in various approaches to program synthesis. Then, several near-term commercial applications of KBSE technology are predicted for the next decade. To scale up from these near-term applications to revolutionizing the entire software life cycle in the next century, the computational requirements of KBSE technology need to be addressed. An examination of current benchmarks reveals that hardware performance by the year 2000 is not likely to be a limiting factor but that fundamental issues such as search control require further research. Finally, the future of KBSE in the next century is presented from the viewpoint of different people in the software life cycle—from end users to the knowledge engineers who encode domain knowledge and design knowledge in software architectures.

where given a domain theory, it will be faster to interactively develop a program with one of these systems than by hand. A key to this breakthrough will be continued improvements in search control. Substantial research programs are now under way to scale KBSE technology up from programming in the small to programming in the large.

where given a domain theory, it will be faster to interactively develop a program with one of these systems than by hand. A key to this breakthrough will be continued improvements in search control. Substantial research programs are now under way to scale KBSE technology up from programming in the small to programming in the large.

The Next Century

Software engineering will evolve into a radically changed discipline. Software will become adaptive and self-configuring, enabling end users to specify, modify, and maintain their own software within restricted contexts. Software engineers will deliver knowledge-based application generators rather than unmodifiable application programs. These generators will enable an end user to interactively specify requirements in domain-oriented terms, as is now done by

...the current generation of CASE tools is limited by shallow representations and...reasoning methods.

telephone engineers with WATSON, and then automatically generate efficient code that implements these requirements. In essence, software engineers will deliver the knowledge for generating software rather than the software itself.

Although end users will communicate with these software generators in domain-oriented terms, the foundation for the technology will be formal representations. Formal representations can be viewed as the extension of current CASE representations, which only capture structural and syntactic information about a software design, into complete semantic representations that capture the full spectrum of software design knowledge. Formal languages will become the lingua franca, enabling knowledge-based components to be composed into larger systems. Formal specifications will be the interface between interactive problem-acquisition components and automatic program synthesis components.

Software development will evolve from an art into a true engineering discipline. Software systems will no longer be developed by handcrafting large bodies of code. Rather, as in other engineering disciplines, components will be combined and specialized through a chain of value-added enhancements. The final specializations will be done by the end user. KBSE will not replace the human software engineer; rather, it will provide the means for leveraging human expertise and knowledge through automated reuse. New subdisciplines, such as domain analysis and design analysis, will emerge to formalize knowledge for use in KBSE components.

Capsule History of Software Engineering

Since the introduction of the electronic digital computer at the end of World War II, hardware performance has increased by an order of magnitude every decade. This rate of improvement has accelerated with the recent introduction of reduced instruction set computer (RISC) architectures, which simplify hardware by pushing complex instructions into software to optimize performance and

shorten design times. As hardware has become less expensive, more resources have been devoted to making computers easier to use and program. User interfaces have evolved from punched cards for batch processing to teletypes and cathode ray tubes for time sharing to graphic user interfaces for networked workstations. In the nineties, new modes of interaction such as handwriting and voice will become common.

Likewise, computers are becoming easier to program. In the fifties and sixties, the first "automatic programming" tools were introduced—assemblers and compilers. Research in programming languages and compiler technology has been a major success for computer science, raising the level of programming from the machine level toward the specification level. In the seventies, interactive programming environments such as Interlisp were created that enabled programs to be developed in small increments and made semantic information about a software system readily available to the programmer (Barstow, Shrobe, and Sandewall 1984). In the eighties, languages designed to facilitate the reuse of software components were introduced, such as Ada and object-oriented extensions of C. Object-oriented programming methodologies encourage a top-down approach in which general object classes are incrementally specialized.

As hardware performance increased, the scope of software projects soon exceeded the capabilities of small teams of programmers. Coordination and communication became dominant management concerns, both horizontally across different teams of programmers and vertically across different phases of software development. Structured methods were introduced to manage and guide software development. In the late sixties and early seventies, structured programming was introduced for incrementally developing correct code from specifications (Dahl, Dijkstra, and Hoare 1972). In *structured programming*, a specification is first developed that states the intended function of a program, and then an implementation is developed through iterative refinement. Structured programming also prescribes methods for making maintenance

easier, such as block-structuring programs to simplify control flow.

Although structured programming helped to correct the coding errors of the sixties, it did not address requirement analysis errors and system design errors. These errors are more difficult to detect during testing than coding errors and can be much more expensive to fix. To address this problem, in the late seventies, structured methods were extended to structured analysis and structured design (Bergland and Gordon 1981; Freeman and Wasserman 1983). These methods prescribe step-by-step processes for analyzing requirements and designing a software system. The manual overhead in creating diagrams and documentation was one limitation of structured methods. A more fundamental limitation is that they provide only limited means for validating analysis and designs and, thus, work best for developing familiar types of software, as is common in commercial data processing. For new types of software, exploratory programming techniques developed in the AI community provide better means for incrementally developing and validating analysis and designs.

CASE was introduced in the mid-eighties to provide computer support for structured methods of software development (Chikofsky 1989; Gane 1990). CASE tools include interactive graphic editors for creating annotated structure chart, data flow, and control-flow diagrams. Like CAD, the development of graphically oriented personal computers and workstations has made CASE economically feasible. The information in these diagrams is stored in a database called a *repository*, which helps to coordinate a software project over the whole development life cycle. In integrated CASE environments, project management tools use the repository to help managers decompose a software project into subtasks, allocate a budget, and monitor the progress of software development.

CASE technology includes limited forms of automatic code generation. Module and data declarations are generated directly from information in a repository. In addition, special-purpose interpreters or compilers called *application generators* have been developed for stereotyped software such as payroll programs and screen generators for video display terminals. Application generators are essentially a user-friendly front end with a back-end interpreter or sets of macros for code generation. In contrast to knowledge-based program synthesis systems, current application generators have narrow coverage, are difficult to modify, are not composable, and provide limited

semantic processing. Nonetheless, they have been successful; it is estimated that over half of current COBOL code is developed by application generators.

The CASE market for just commercial data processing tools totaled several billion dollars in 1988 and is doubling every three to four years (Schindler 1990). The major advantages of using current CASE technology are that it enforces a disciplined, top-down methodology for software design and provides a central repository for information such as module interfaces and data declarations. However, current CASE technology can only represent a small portion of the design decisions necessary to build, maintain, and modify a software system. CASE design tools mainly represent the structural organization of a software design, not the function of the components of a software system. Current CASE tools only provide limited forms of analysis, such as checking the integrity of control and data flow and the consistency of data declarations across modules. CASE technology does not currently provide a means for validating the functional behavior of a software design to ensure that the design satisfies the needs of the customer.

AI and Software Engineering Today

AI has already made a significant impact on software engineering. First, mainstream software engineering has adopted AI programming techniques and environments, including expert system shells, as prototyping tools. A *software prototype* is a system constructed for evaluation purposes that has only limited function and performance. Second, AI components are being integrated into larger systems, particularly where flexibility and ease of modification are needed for rapidly changing requirements. Third, AI inference technology is providing the foundation for more powerful and user-friendly information systems (Barr 1990). Fourth, AI programming paradigms are being adopted in more conventional environments, including graphic user interfaces, object-oriented programming, constraint-based programming, and rule-based programming. In all these applications, the major factor has been the ease of developing, modifying, and maintaining programs written in AI programming environments.

AI technology provides particularly effective exploratory programming tools for poorly understood domains and requirements (Shiel 1986). Exploratory programming converges on well-defined requirements

and system specifications by developing a prototype system, testing the prototype with the end user to decide whether it satisfies the customer's needs, and then iteratively modifying the prototype until the end user is satisfied. Exploratory programming identifies errors in requirements and specifications early in the design process, when they are cheap to fix, rather than after the system has been delivered to the customer, when they can cost a hundred to a thousand times as much to fix. This advantage of early feedback with a prototype system is leading to the replacement of linear methods of software development with methods that incorporate one or more passes of prototype development. AI technology is suitable for exploratory programming because its programming constructs, such as objects, rules, and constraints, are much closer to the conceptual level than conventional programming constructs. This approach enables prototype systems to be rapidly constructed and modified.

Prototype systems written in very high-level languages and environments can directly be evolved into working systems when efficient performance is not necessary. For example, many small businesses build their own information and accounting systems on top of standard database and spreadsheet programs. Historically, spreadsheet programs descended from VISICALC, a simple constraint-based reasoning system within a standard accounting paradigm. For scientific applications, Wolfram's MATHEMATICA environment enables scientists and engineers to rapidly develop mathematical models, execute the models, and then graph the results. MATHEMATICA uses a programmable rule-based method for manipulating symbolic mathematics that can also be used for transformational program derivations.

However, when efficient performance is necessary, the current generation of high-level development environments does not provide adequate capabilities because the environments interpret or provide default translations of high-level programs. Although in the future, faster hardware can compensate for constant factor overheads, the most difficult inefficiencies to eliminate are exponential factors that result from applying generic interpretation algorithms to declarative specifications. For example, in the mid-eighties, the New Jersey motor vehicle department developed a new information system using a fourth-generation language, which is essentially an environment for producing database applications. Although the fourth-generation language enabled the

system to be developed comparatively fast, the inefficiency of the resulting code caused the system to grind to a halt when it came online and created havoc for several years for the Garden State. As another example, although mathematical models of three-dimensional physical systems can be specified with MATHEMATICA, the execution of large or complex models can be unworkably slow.

When efficient performance is necessary, a prototype system currently can only be used as a specification for a production system; the production system has to be coded manually for efficiency. For example, after a three-dimensional physical system is specified in MATHEMATICA, it still needs to be coded into efficient Fortran code to run simulations. Although both MATHEMATICA and MACSYMA can produce default Fortran code from mathematical specifications, the code is not efficient enough for simulating large three-dimensional systems.

The necessity of recoding a prototype for efficiency incurs additional costs during development but has even more pernicious effects during the maintenance phase of the software life cycle. As the production system is enhanced and modified, the original prototype and documentation are seldom maintained and, therefore, become outdated. Also, because modifications are done at the code level, the system loses its coherency and becomes brittle. Eventually, the system becomes unmaintainable, as described in chapter 1 of *Automating Software Design*.

Transformational Programming

KBSE seeks to combine the development advantages of high-level environments supporting rapid prototyping with the efficiency advantages of manually coded software. The objective is to create a new paradigm for software development—transformational programming—in which software is developed, modified, and maintained at the specification level and then automatically transformed into production-quality software (Green et al. 1986). For example, the SINAPSE system automates the production of efficient Fortran code from high-level specifications of three-dimensional mathematical models. As another example, the ELF system automates the production of efficient, VLSI wire routing software from specifications. Where this automatic translation is achieved, software development, maintenance, and modification can be carried out at the specification level with the aid of knowledge-based tools. Eventually, software engineering will evolve to a higher level



CASE was introduced...to provide computer support for structured methods of software development...

and become the discipline of capturing and automating currently undocumented domain and design knowledge.

To understand the impact of automating the translation between the specification level and the implementation level, consider the impact of desktop publishing during the past decade. Before the introduction of computer-based word processing, manually typing or typesetting a report consumed a small fraction of the time and money needed to generate the report. Similarly, manually coding a software system from a detailed specification consumes a small fraction of the resources in the software life cycle. However, in both cases, this manual process is an error-prone bottleneck that prevents modification and reuse. Once a report is typed, seemingly minor modifications, such as inserting a paragraph, can cause a ripple effect requiring a cascade of cutting and repasting. Modifying a software system by patching the code is similar to modifying a report by erasing and typing in changes. After a few rounds of modification, the software system resembles an inner tube that is patched so often that another patch causes it to blow up. When a typewritten report gets to this stage, it is simply retyped. However, when a software system gets to this stage, the original design information is usually lost. Typically, the original programming team has long since departed, and the documentation has not been maintained, making it inadequate and outdated. The only recourse is an expensive reengineering effort that includes recovering the design of the existing system.

Because maintenance and modification are currently done at the code level, they consume over half the resources in the software life cycle, even though the original coding consumes a small fraction of the life-cycle resources. Most maintenance effort is devoted to understanding the design of the current system and understanding the impact of proposed modifications. Furthermore, because modification is difficult, so is reuse. It is easy to reuse portions of previous reports when they can easily be modified to fit in a new context. Similarly, it is easy to reuse portions of previous software systems when abstract components can easily be adapted to the con-

text of a new software system. For these reasons, word processing and desktop publishing have had an impact disproportionate to the resources consumed by the manual process they automate. For similar reasons, automating the development of production-quality software from high-level specifications and prototypes will have a revolutionary impact on the software life cycle.

By automating printing, desktop publishing has created a market for computer-based tools to help authors create publications from the initial conceptual stages through the final layout stage. Outlining programs help authors organize and reorganize their ideas. Spelling checkers and grammar checkers help ensure consistency with standards. Page layout programs optimize the visual presentation of the final publication and often integrate several source files into a final product. These tools would be useful even in the absence of automated printing. However, they reach their full potential in environments supporting the complete spectrum of activities for the incremental and iterative development of publications.

Current CASE tools are similar to outliners, grammar checkers, and tools that integrate several source files. CASE tools have not yet reached their full potential because coding is still mostly a labor-intensive manual process. As coding is increasingly automated, more sophisticated tools that support the initial stages of software design will become more useful. Rapidly prototyped systems developed with these tools will be converted with minimal human guidance into production-quality code by program synthesis systems, just as word processing output today is converted almost automatically into publication-quality presentations through page layout programs. CASE representations will move from partial representations of the structure and organization of a software design toward formal specifications. Tools such as WATSON that interactively elicit requirements from end users and convert them into formal specifications will be one source of formal specifications. Tools such as ARIES will enable software developers to incrementally develop and modify formal specifications. Tools such

as ROSE-2 will match specifications to existing designs for reuse.

Comparison of Program Synthesis Techniques

The ultimate objective of transformational programming is to enable end users to describe, modify, and maintain a software system in terms natural to their application domain and, at the same time, obtain the efficiency of carefully coded machine-level programs. Great progress has been made toward this objective in the progression from assemblers to compilers to application generators and fourth-generation languages. However, much remains to be done.

Given current technological capabilities, there are trade-offs between several dimensions of automatic translation. The first dimension is the distance spanned between the specification or programming level and the implementation level. The second dimension is the breadth of the domain covered at the specification level. The third dimension is the efficiency of the implemented code. The fourth dimension is the efficiency and degree of automation of the translation process. The fifth dimension is the correctness of the implemented code. The *Automating Software Design* chapters in the sections on domain-specific program synthesis, knowledge compilation, and formal derivation systems show how current KBSE research is expanding our capabilities along each of these dimensions. The following paragraphs compare different approaches to automatic translation.

Compilers for conventional programming languages such as Fortran and COBOL have a wide breadth of coverage and efficient translation and produce fairly efficient code. These accomplishments are achieved with a comparatively short distance between the programming level and the implementation level. Optimizing compilers sometimes have user-supplied pragmas for directing the translation process, thereby getting greater efficiency in the implemented code at the expense of greater user guidance in the translation process. Verifying that a compiler produces correct code is still a major research issue.

Very high-level languages such as logic programming and knowledge interpretation languages also achieve a wide breadth of coverage through a short distance between the programming level and the implementation level. However, the programming level is much closer to the specification level than with conventional programming languages.

Programs written in these languages are either interpreted or compiled to remove interpretive overhead such as pattern matching. These languages can be used to write declarative specifications that are generally implemented as generate-and-test algorithms, with correspondingly poor performance. It is also possible to develop efficient logic programs by tuning them to the operation of the interpreter, in effect, declaratively embedding control structure. The code implemented for these programs approaches, within a constant factor, the efficiency of code written for conventional languages, except for the lack of efficient data structures. Efficient logic programs can either be developed manually or can be the target code of a program synthesis system such as PAL OF XPRTS

Knowledge interpreters such as expert system shells are seldom based on formal semantics, and thus, formally verifying that they produce correct code is impossible. However, some research-oriented languages such as pure Prolog are based on formal semantics. Formal proofs in which abstract compilers for these languages produce correct code for abstract virtual machines have appeared in the research literature (Warren 1977; Despeyroux 1986). These proofs are based on the close relationship between logic programming and theorem proving, for which the soundness and completeness of inference procedures have mathematically been worked out. Proving that real compilers produce correct code for real machines is a much more difficult and detailed undertaking. Success has recently been reported in Hunt (1989), Moore (1989), and Young (1989).

Application generators span a significant distance between the specification level and the implementation level by choosing a narrow domain of coverage. Application generators typically use simple techniques to interpret or produce code in conventional languages, such as filling in parameters of code templates. Thus, the performance of implemented code usually is only fair. The translation process is efficient and automatic. Because application generators are mainly used in commercial data processing, mathematically verifying that an application generator produces correct code has not been a major issue.

Domain-specific program synthesis systems will likely become the next generation of application generators. They also span a significant distance between the specification level and the implementation level by choosing a narrow domain of coverage. They are easier to incrementally develop and modify than conventional application generators

*The objective
is to create a
new paradigm
for software
development
—transformational
programming...*

because they are based on transformation rules. It is also easier to incorporate more sophisticated problem-solving capabilities. Therefore, compared to conventional application generators, they tend to produce much better code and provide higher-level specifications. In theory, the transformation rules could be derived by composing basic transformation rules that were rigorously based on logic and, therefore, could verifiably be correct. However, in practice, the transformation rules are derived through knowledge engineering. Compared with other program synthesis approaches, the narrow domain enables domain knowledge and search knowledge to be hard wired into the transformation rules; therefore, the translation process is comparatively efficient.

Knowledge compilation research seeks to combine the high-level specification advantages of knowledge interpreters with the efficiency advantages of conventional programs. The goals are broad coverage, a large distance between the specification level and the implementation level, and efficient code. To achieve these goals, the major research issue is search control during program synthesis. Like approaches to domain-specific synthesis, the transformation rules are derived through knowledge engineering and, hence, are not verifiably correct. However, the domain knowledge is usually encoded separately and explicitly; therefore, knowledge compilers can work in different domains given different domain knowledge.

Formal derivation research seeks the same goals as knowledge compilation research but within a framework that is rigorously based on logic and, therefore, is verifiably correct. The formal basis for the extraction of programs from constructive proofs and for basic transformation rules was worked out 20 years ago. As in knowledge compilation, the major research issue is search control during program synthesis. The major approach to search control is to develop metalevel programs called *tactics* and *strategies* that encapsulate search control and design knowledge.

The Next Decade

KBSE will revolutionize the software life cycle, but the path will be evolutionary because its development, as well as its incorporation into software-engineering practice, will be incremental. KBSE will adapt itself to current practices before it changes these practices.

In the coming decade, there are several leverage points where KBSE technology could commercially be applied. In each of these

potential uses, technology already developed in the research laboratory could be combined with conventional software-engineering tools to provide much better tools. For some of these applications, industrial pilot projects are already under way. First, I review several potential near-term uses, then examine applications in software maintenance and special-purpose program synthesis. The computational requirements of KBSE technology are also discussed.

First, KBSE tools for software understanding will likely be adopted for software maintenance (as described earlier). Current AI technology can significantly enhance standard software understanding tools such as data flow and control-flow analyzers. KBSE aims at eventually elevating maintenance from the program level to the specification level. However, because of the enormous investment in the existing stock of software, program maintenance will be the dominant cost in software engineering for decades to come.

Second, AI technology could be used to produce the next generation of application generators (see *The Next Century*). Transformational programming provides a flexible and modular basis for a deeper level of semantic processing than is feasible with current application generators. Its use leads to better user interaction and higher-level specifications as input and more optimal code as output.

Third, AI environments for rapid prototyping and exploratory programming could be enhanced and integrated with CASE design tools. Rapid prototyping has recently become a major topic in software engineering. CASE design tools support decomposing a system into a hierarchy of modules. The result is usually a hierarchical graph representation annotated with text commenting on the intended function of the modules. AI programming environments can be used to rapidly prototype the function of these modules and, thus, create an executable prototype of the whole system.

Fourth, the currently small but growing use of formal methods (Wing 1990) for software development could considerably be enhanced through software-engineering technology. A method is formal if it has a sound mathematical basis and, therefore, provides a systematic rather than ad hoc framework for developing software. In Europe, formal languages such as VDM (Jones 1986) and Z (Spivey 1989) have been used in pilot projects to manually develop formal specifications of real software systems. Tools that incorporate automated reasoning could provide substan-

tial assistance in developing formal specifications, as shown earlier. Furthermore, within this decade, formal derivation systems will become sufficiently advanced to provide interactive environments for refining formal specifications to code (see Transformational Programming).

Finally, a major problem with programming in the large is ensuring that a large software system is consistent: The implementation must be consistent with the system design that, in turn, must be consistent with the requirements; the components of the software system developed by separate teams must be consistent with each other. Current CASE tools use shallow representations and, thus, provide only limited consistency checking. For example, CASE design tools can check that a hierarchy of modules has no dead ends in the flow of control. More complete consistency checking will require deeper representations and more inference capabilities. The research on ROSE-2 shows that there is an evolutionary path from current CASE consistency checking to consistency checking with deeper representations.

Software Maintenance: The Next Decade

Industry and government have invested hundreds of billions of dollars in existing software systems. The process of testing, maintaining, modifying, and renovating these systems consumes over half of the software-engineering resources and will continue to do so for decades to come. Often, older systems are no longer maintainable because no one understands how they work, preventing these systems from being upgraded or moved to higher-performance hardware platforms. This problem is acute for software written in archaic languages that programmers are no longer trained in.

Intelligent software maintenance assistants are based on AI techniques for software understanding and ramification reasoning. Software understanding is a prerequisite to other maintenance tasks and currently accounts for over half the time spent by maintenance programmers. In the long term, software systems will include built-in self-explanation facilities such as those in LASSIE and the explainable expert system (Neches, Swartout, and Moore 1985). Today, software understanding is done by *reverse engineering*, analyzing existing code and documentation to derive an abstract description of a software system and recover design information.

Reverse engineering is the first step in *reengineering*, renovating existing systems, including porting them to newer languages

and hardware platforms. Because maintenance is increasingly difficult, reengineering is especially important for code written long ago in older languages and for older hardware platforms. Many businesses are legally required to be able to access records in databases dating back decades. These databases were written as flat files, making it impossible to integrate them with newer relational databases and requiring the business to keep backward compatibility with old hardware platforms. Reengineering is also needed to port engineering and scientific code written in the sixties and seventies to newer hardware platforms with parallel architectures. Reengineering these older systems occupies many programmers; AI technology can provide significant assistance to this task.

Standard AI techniques such as pattern matching and transformation rules enhance conventional tools for software understanding by enabling higher-level analysis and abstraction (Biggerstaff, Hoskins, and Webster 1989; Hartman 1989; Letovsky 1988; Wills 1989). With these techniques, much of the information needed by current CASE design tools can be recovered semiautomatically, even from archaic assembly language code. One approach is *cliche recognition* (Rich and Waters 1990), matching stereotyped programming patterns to a program's data and control flow, thereby abstracting the program into a hierarchy of these cliches.

AI techniques for software understanding can also be applied to the testing and integration phases of software development. For example, a major aerospace company used a KBSE system to test two million lines of Fortran code produced by subcontractors for compliance with its coding standards. Hundreds of violations were found, including unstructured do loops, dead code, identifier inconsistency, and incorrectly formatted code. This technique has already saved four person-years of hand checking, yet the system took less than half a person-year to develop. The system was built in REFINE, which is a very high-level programming environment that integrates parsing technology, an object-oriented knowledge base, and AI rule-based technology (Burson, Kotik, and Markosian 1990). Most of the development time was devoted to writing grammar definitions for the three dialects of Fortran used by subcontractors; REFINE automatically generates parsers from the grammar definitions. The core of the system was written as a set of rules and took only a few weeks of development time. Because it was written as a set of rules, it is easily modifiable as coding standards change.

KBSE will revolutionize the software life cycle...

Understanding the existing code is the first step in software modification. The next step is ramification reasoning to determine the impact of proposed modifications to a software system. The objective is to ensure that a modification achieves its goal without causing undesired changes in other software behavior. This is essentially a task for constraint-based reasoning. Ramification reasoning can be done at many levels of abstraction. At the lowest level, it involves tracing through data flow and control-flow graphs to decide which programs and subroutines could be affected by a proposed modification. Automated ramification reasoning is even more effective when given higher-level design information. For example, it can determine whether proposed modifications violate data-integrity constraints and other invariants. As CASE representations evolve into more complete and formal representations, AI techniques for ramification reasoning will be incorporated into CASE tools.

Special-Purpose Program Synthesis: The Next Decade

Today, application generators are one of the most effective tools for raising programmer productivity. Because they are based on code templates, they provide a more flexible and higher level of software reuse than the reuse of code. Transformational technology provides the means for higher-performance application generators, which are also potentially easier to develop and modify.

Application generators generally produce code in three phases. The first syntactic analysis phase converts a specification written in an application-oriented language or obtained interactively through menus and forms into a syntactic parse tree. A semantic analysis phase then computes semantic attributes to obtain an augmented semantic tree. The final generation phase traverses the semantic tree and instantiates code templates. The generation phase is similar to macroexpansion in conventional programming languages.

Application generator generators (Cleveland and Kintala 1988) are tools for building application generators. Parser generators such as YACC take the definition of an application-oriented language as input and produce a parser for the syntactic analysis phase as output. Tools for building the semantic analysis phase are usually based on attribute-grammar manipulation routines. Tools for building the generation phase are similar to macro definition languages.

AI provides technology for much richer semantic processing. In addition, the genera-

tion phase can be augmented with program transformations to produce better code. Program transformations enable code templates to be more abstract and, therefore, have wider coverage. For example, code templates can be written with abstract data types, such as sets, that program transformations, then refine into concrete data structures.

Transformation rules provide a flexible and modular basis for developing and modifying program transformations and semantic analysis routines. Eventually, end users will be able to interactively develop their own transformation rules in restricted contexts. In the near term, transformational technology will enable software engineers to build higher-performance application generators. KBSE environments that include parser-printer generators and support for program-transformation rules, such as REFINE, provide tools for developing knowledge-based application generators.

Intelligent application generators are cost-effective options not only in standard application areas such as business information systems but also in the enhancement of existing software. Automatic code enhancement makes software development and maintenance more efficient and results in fewer bugs. Examples of enhancements include better error handling and better user interfaces.

Making software fault tolerant is particularly important in real-time systems and in transaction systems where the integrity of a database could be affected. Consequently, a significant amount of code in these systems is devoted to error detection and recovery. An informal sampling of large telecommunications programs found that 40 percent to 80 percent of branch points were devoted to error detection.

User interface code often accounts for 30 percent of an interactive system. Screen generators have long existed for business transaction systems, and graphic user interface generators for workstations have recently been marketed. However, current tools only make it faster to develop precanned displays. Intelligent interfaces are much more flexible. They can adapt to both the semantics of the data they present and the preferences of the user. For example, instead of precanned text, intelligent interfaces use natural language generation techniques that are sensitive to a model of the user's knowledge. Graphics can be optimized to emphasize information important to the user.

However, intelligent interfaces are more difficult to design and program than standard

user interfaces, making it costly to incorporate them in each new application. Roth, Mattis, and Mesnard (1990) describe *SAGE*, a prototype system for application-independent intelligent data presentation. *SAGE* incorporates design expertise for selecting and synthesizing graphic components in coordination with the generation of natural language descriptions. *SAGE* combines a declarative representation of data with knowledge representations of users' informational goals to generate graphics and text. An application developer can generate an intelligent presentation system for his(her) application using *SAGE*.

Program transformation technology can directly be applied to the synthesis of visual presentations. Westfold and Green (1991) describe a transformation system for deriving visual presentations of data relationships. The underlying idea is that designing a visual presentation of data relationships can be viewed as designing a data structure. Starting with a relational description of data, transformations are applied to reformulate the description into equivalent descriptions that have different implementations, that is, different representations on the screen. Many different representations of the same data can be generated, each emphasizing different aspects of the information. For example, an n -ary relation can be reformulated so that all the tuples with the same first argument are grouped. Successive applications of this rule gradually transform the n -ary relation into a tree. Other rules transform data relationships into display-oriented relationships, which are then rendered graphically.

Computational Requirements

KBSE can be expensive computationally, both in terms of memory and processor cycles. The research systems described in this book often push the performance envelope of current workstations. Hardware capabilities have previously been a limiting factor in the commercial adoption of CAD, CASE, and AI technology. The most limiting factor was the computational requirements of providing a graphic user interface with the hardware available in the late seventies and early eighties. Hardware performance in the nineties will probably not be a major limiting factor in scaling up KBSE technology to industrial applications.

Benchmarks from several industrial pilot projects show that current computer workstations provide sufficient computational power to support KBSE on real problems when appropriate trade-offs are made. A benchmark from domain-specific program synthesis is the

SINAPSE system, which synthesizes a 5,000-line Fortran three-dimensional modeling program from a 60-line specification in 10 minutes on a SUN-4 (a SUN-4 is a UNIX workstation running about 12 MIPS with memory from 16 to 64 megabytes). This type of program would take weeks to generate by hand. SINAPSE is specialized to synthesize only finite-difference algorithms, so it can make large-grained decisions without sacrificing automation or the efficiency of the resulting code. SINAPSE also uses knowledge-based techniques to constrain the search space. Another benchmark comes from the testing of Fortran code for adherence to coding standards, which was described earlier. On a SUN-4, 20,000 lines of code were checked each hour.

The computational demands of KBSE technology result from the scope of the knowledge representation, the granularity of automated decision making, and the size and complexity of the software systems produced. Greater breadth or depth of representation requires increased memory, and increased granularity of automated decision making requires increased processor cycles. The critical issue is how computational requirements scale as a function of these variables.

The computational requirements of software-engineering technology can be factored into two groups: those that increase either linearly or within a small polynomial in terms of these variables and those that increase exponentially. The former group includes factors such as the amount of memory needed to represent the history of a derivation, including all the goals and subgoals. As discussed in chapter 23 of *Automating Software Design*, this amount is probably proportional to $N * \log N$, where N is the size of the derivation. The exponential factors include combinatorially explosive search spaces for program derivations. These exponential factors cannot be addressed by any foreseeable increase in hardware performance. Instead, better KBSE technology is required, such as tactics for search control.

Although software engineers will always want more hardware power, increases in hardware performance within the next decade will likely address the computational requirements of KBSE that do not grow exponentially. The benchmarks from industrial pilot projects show that some KBSE technology can already be applied to industrial-scale software systems within the performance limitations of current computer workstations.

Hardware performance within the fiercely competitive workstation market will continue to grow rapidly. Based on technology already

KBSE can be expensive... in terms of memory and processor cycles.

In the future, software systems will include built-in, knowledge-based software information systems...

in advanced development, an order of magnitude increase is expected by the middle of the decade in both processor speed and memory size. Simpler architectures such as RISC have decreased development time and will facilitate the introduction of faster device technologies such as gallium arsenide and Josephson junctions.

Lucrative, computationally intensive applications, such as real-time digital video processing, will drive hardware performance even higher in the latter part of the decade, with yet another order-of-magnitude increase. Compared with the computational requirements of these applications, scaling up KBSE technology to industrial applications will not be hardware limited if exponential factors can be avoided. The next part of this article describes how combinatorially explosive search might be avoided through the reuse of domain and design knowledge.

The Next Century

Within the KBSE community, there is a broad consensus that knowledge-based methods will lead to fundamentally new roles in the software-engineering life cycle and a revised view of software as human knowledge that is encapsulated and represented in machine manipulable form. At the beginning of the computer age, this knowledge was represented as the strings of 1's and 0's of machine language. By applying software engineering to itself by developing compilers, the representation level has been raised to data structures and control structures that are closer to the conceptual level. Although this improvement is considerable, most of the domain and design knowledge that is used in developing a modern software system is lost or, at best, is encoded as text in documentation.

The goal of KBSE research is to capture this lost knowledge by developing knowledge representations and automated reasoning tools. As this task is accomplished, software engineering will be elevated to a higher plane that emphasizes formalizing and encoding domain and design knowledge and then automatically replaying and compiling this knowledge to develop working software systems. Below, I envision how future KBSE environments might support different classes of people in the software life cycle, from end users to the knowledge engineers who encode domain knowledge and design knowledge in software architectures.

Maintenance Programmers

In the future, software systems will include built-in, knowledge-based software informa-

tion systems such as LASSIE. Maintenance programmers will no longer study reams of source code and outdated documentation to understand a software system. Instead, they will query the system itself. Eventually, the information will automatically be updated, so that as a system is modified, the information is kept current. The information in these systems will expand to include the full range of design decisions made in developing the software system. Eventually, these software information systems will be produced as a by-product of software development with KBSE tools.

These software information systems will evolve to become active assistants in software maintenance. They will be able to trace the ramifications of proposed changes at the code, system design, and requirement levels. They will help to ensure that as a software system evolves to meet changing requirements, it remains consistent and bug free. Eventually, maintenance will no longer be done by modifying source code. Instead, desired changes will be specified directly by the end user at the requirement level, and the system will carry them out automatically. Software systems will become self-documenting and self-modifying. The typically junior programmers who are now burdened with maintaining code designed by other programmers will spend their time in more interesting pursuits.

End Users

In the future, end users will interactively customize generic software systems to create applications tailored to their own particular needs. Progress in this direction has already been made. For example, several word processing and spreadsheet programs allow users to customize menus, change keyboard bindings, and create macros. Sometimes, macros are compiled for greater efficiency. KBSE technology will provide much more flexibility in customization, more optimized code, and intelligent assistance in helping an end user develop requirements.

A scenario for future intelligent assistance for end users in scientific computing is presented by Abelson et al. (1989). Below, I examine a simpler domain by considering how a businessperson will develop a custom payroll system in the future. Current payroll program generators automate code generation but do not provide substantial help in eliciting and analyzing requirements. An intelligent payroll application generator will include a knowledge base about the payroll

domain that defines concepts such as hourly versus salaried employees, various types of bonuses and deductions, tax codes, and different kinds of pay periods. Using this knowledge, a requirement acquisition component will interactively elicit the relevant structure of the businessperson's company and the types of output and information desired. During this interaction, the requirement acquisition component will check the consistency of evolving requirements with constraints, such as tax codes. It will also guide the businessperson in resolving ambiguities and incompleteness. The result of this requirement elicitation process will be a formal specification of the desired payroll system in the form of declarations and constraints formulated in terms of the concepts and operations of the payroll domain.

To validate the specification, the payroll application generator will then transform these declarations and constraints into a prototype spreadsheet program and try out a few test examples with the businessperson. This step will lead to further refinements of the requirements until the businessperson is satisfied. The payroll program generator will then compile the specification into machine code and generate the proper interfaces for other accounting systems.

The businessperson does not have to be an expert in accounting, tax codes, or business programming or even be able to develop spreadsheets. The domain knowledge of payrolls would enable the requirement acquisition component to elicit the appropriate information in terms the businessperson can understand.

Many of the capabilities described in this sketch are within the range of current KBSE research systems. For example, eliciting payroll requirements involves temporal reasoning that is much less complex than that performed by WATSON in eliciting requirements for new telephone features. Other recent work on knowledge-based requirement acquisition includes Rubenstein and Waters (1989) and Anderson and Fickas (1989). Similarly, compiling the arithmetic constraints of a payroll specification into efficient code is far less difficult than compiling partial differential equations into efficient finite-difference programs, as done by SINAPSE. What is missing are good tools for acquiring the knowledge of the payroll domain, so that this knowledge can be used by both requirement elicitation components and program synthesis components.

System Developers

The needs of system developers differ from those of end users. End users prefer to interact

at the requirement level and be shielded from the complexity of specifications and system design. In contrast, system developers prefer help managing the complexity of specifications and system designs but want to be shielded from the implementation details. Two inter-related ideas currently being developed could meet some of the needs of future system developers: megaprogramming and software architectures.

Megaprogramming is programming at the component level (e.g., user interfaces, databases, device controllers) rather than the code level. To program at the component level, it is necessary to either reuse components or generate components from specifications. Components can more readily be reused or generated if they are defined in the context of a software architecture that identifies major types of components and provides the glue for composing them together. A *software architecture* is a high-level description of a generic type of software system, such as the class of transaction-processing systems.

Software architectures will subsume current CASE design representations. To support software developers, software architectures will include the functional roles of major software components and their interrelationships stated in an application-oriented language; a domain theory that provides precise semantics for the application-oriented language for use in automated reasoning; libraries of prototype components with executable specifications; program synthesis capability to produce optimized code for components after a prototype system has been validated; a constraint system for reasoning about the consistency of a developing software system; and design records that link requirements to design decisions, as in the ROSE-2 system .

In the future, routine system development will be done by small teams of system analysts and domain experts working with KBSE environments that support software architectures. In contrast to the guidance provided for an end user, the KBSE environment will play an assistant role in requirement analysis. The team will start with a set of informal requirements and elaborate them into precise statements in the application-oriented language. As the requirements are made precise, the software-engineering environment will propagate these decisions through the design records to check the feasibility of meeting these requirements. This propagation will also set up default design decisions and record design alternatives to be investigated

Domain knowledge is a prerequisite to requirement engineering.

by the team.

After a subset of the requirements has been made into precise specifications, the KBSE environment will generate executable prototypes to help validate and refine the requirements. Analytic assessments of an evolving design will be provided by having the constraint system determine the ramifications of design decisions. The constraint system will ensure that design decisions are internally consistent and consistent with the software architecture. The team will iteratively refine its requirements, the precise specifications, and the system design using an opportunistic methodology similar to current rapid prototyping methodologies. After the system design is complete, the implementation of the production system will mostly be automatic. The KBSE environment will ask for parameters relevant to performance, such as the size of expected input, and will occasionally ask for guidance on implementation decisions.

System development by a close-knit team with automated support will diminish many communication, coordination, and project management difficulties inherent in managing large numbers of programmers. The feasibility of small teams developing prototype systems has already been shown using an AI programming environment suitable for rapid prototyping, reusable prototype components, and a software architecture (Brown et al. 1988). However, implementing the final production system required a large number of programmers to code efficient versions of the components. In the future, this final production phase will largely be automated.

There is an inherent tension between reusability and performance in component implementations, which is why megaprogramming is currently easier to apply to system prototyping than production system implementation. To be maximally reusable, components should make as few assumptions about other components as possible. Reusable components should use abstract data types and late-binding mechanisms common in AI environments. To be maximally efficient, components should take advantage of as much of the context of use as possible, including the implementations of data types in other components and early binding mechanisms. Transformational programming will make it possible to use megaprogramming during system prototyping and then automatically generate efficient production-quality code.

Extensible software and additive programming are the foundation for enabling small

teams to build large software systems, end users to customize their own applications, and application software to be self-maintaining. A software architecture represents a partial set of decisions about requirements and system design that are generic to a class of software systems. To be reusable, this partial information must be represented so it is extensible. System developers and end users will incrementally add to these decisions to generate a software system. Systems will be modified for changing requirements by retracting decisions and adding new decisions.

Software Architects, Domain Analysts, and Design Analysts

Software architectures will be created through domain analysis and design analysis. *Domain analysis* (Arango 1988) is a form of knowledge acquisition in which the concepts and goals of an application domain are analyzed and then formalized in an application-oriented language suitable for expressing software specifications. *Design analysis* is the formalization of transformational implementations for a class of software artifacts. A *domain designer* is a design analyst who derives implementations for the objects and operations in an application-oriented language developed through domain analysis.

System developers will reuse domain analysis by stating requirements, specifications, and system designs in the application-oriented language. System developers will also reuse design analysis when they develop an application system through a software architecture. Thus, the cost of domain analysis and design analysis will be spread over many different systems.

Domain knowledge is necessary for intelligent requirement analysis, specification acquisition, and program synthesis. Domain knowledge can implicitly be embedded in special-purpose rules or can be formal and explicit. Formalizing domain knowledge is a difficult task, currently requiring extended collaboration between domain experts and knowledge engineers (Curtis, Krasner, and Iscoe 1988). One advantage of formalizing domain knowledge is that it can then be used in all phases of software development, with the assurance of correctness between implementations and specifications. In other words, if a user validates a specification with a specification assistant, and a program synthesis system derives an implementation for this specification, the program correctly implements the user's intentions. Formalized domain knowledge can also be used with generic automated reasoning tools, thus

enabling the reuse of KBSE components.

Domain knowledge is a prerequisite to requirement engineering. Requirements are necessarily described in terms of the application domain in which a software system will be used. Current requirement languages are restricted to expressing system-oriented concepts. Knowledge representation languages will provide the basis for expressing domain knowledge within which domain-oriented requirements can formally be expressed (Borgida, Greenspan, and Mylopoulos 1986). This approach will enable automatic theorem provers to verify whether a set of requirements is consistent with domain knowledge and to fill incompleteness in a partial set of requirements, as is done by WATSON for the telephone domain. Program synthesis systems use domain knowledge to decompose and implement domain objects and operations. For example, distributive laws in a domain theory are used to decompose and optimize domain operations.

A future design analyst will first interactively develop transformational derivations for a set of generic programs in an application domain using a general-purpose program synthesis system. These transformational derivations will be recorded as derivation histories. *Derivation histories* can be viewed as the execution trace of an implicit metalevel program that controls the transformational derivation. The objective of design analysis is to make at least part of the information in this metalevel program explicit so that it can be stored in a software architecture and reused by a system developer. A derivation history needs to be generalized so that it can be applied to similar but not necessarily identical specifications. Because of the difficulty of automatically generalizing execution traces to programs, I anticipate that generalizing derivation histories will be a manual process, with some computer assistance, for the foreseeable future. In the long-term future, this generalization process might be automated through explanation-based generalization, given a general theory of the teleological structure of transformational derivations. Applications of explanation-based generalization to program synthesis are described in chapters 14 and 22 of *Automating Software Design* and also in Shavlik (1990). Fickas (1985) describes GLITTER, a research system that uses teleological structure to partially automate transformational derivations.

The first level of generalizing a derivation history will be done by annotating the derivation history with the dependencies between the individual transformation steps and the

overall goal structure of the derivation. Some dependency information will automatically be generated. The annotations will provide information to intelligent replay systems to modify a derivation history for use in similar specifications (Mostow 1989). The second level of generalization will be done by creating metalevel tactics. A third level of generalization will be strategies that encapsulate heuristic knowledge for choosing tactics. These strategies will include methods for decomposing specifications into components to be synthesized by specialized tactics and methods for using performance requirements to guide the transformational derivation.

A software architect will integrate the results of domain analysis and design analysis with a constraint system for reasoning about the consistency of a developing software system. The results of domain analysis will be an application-oriented language for a software architecture and a domain theory that defines the semantics of this language. The results of design analysis, that is, annotated derivation histories, tactics, and strategies, will essentially form a set of special-purpose program synthesizers for the components of a software architecture. The software architect will also provide an initial set of design records linking requirements to design decisions for choosing one component over another. These design records will be elaborated by teams of system developers.

In essence, software engineering will become knowledge acquisition followed by redesign. Although creating a software architecture will require more effort than designing any particular software system, it will be paid back over the creation of many software systems. Software architectures will provide the right tools for what is now partially done through copy and edit. Instead of designing a system from scratch, software system developers will extend and redesign the defaults in a software architecture. *Redesign* is the method used for iteratively modifying a rapidly prototyped system. In the future, redesign will be supported with a spectrum of tools, including specification evolution transformations, consistency maintenance systems, and intelligent replay of derivation histories.

KBSE environments that support domain analysis, design analysis, and software architectures are being explored in research laboratories. Domain analysis and design analysis are knowledge-acquisition tasks. A prototype domain analysis environment is DRACO, which includes generators for application-oriented languages. DRACO also provides support for transformation rules within a domain and

between domains. KIDS and WATSON provide some support for creating and maintaining formal domain theories. In the future, knowledge-acquisition tools will provide sophisticated environments for domain analysis (Marcus 1989).

Design analysis is the acquisition of control knowledge. Representing and acquiring this knowledge is critically dependent on having a good understanding of the structure of transformational derivations. As shown by the chapters in *Automating Software Design*, great progress has been made since the early days of program synthesis. For the most part, the program derivations presented here are at a higher level of abstraction than the details of the logical calculi that underlay the individual transformations. Several of the interactive program transformation systems, especially KIDS, provide a natural high-level interface for those schooled in transformational derivations. However, we are just beginning to understand how to encode the rationale for choosing particular paths through the design space. This semantic information is what we need for generalizing derivation histories and developing tactics and strategies. I expect significant progress over the coming decade. For a comparative study of different derivations in the program synthesis literature, see Steier and Anderson (1989), especially the concluding chapter on design space.

Summary

To date, the main use of AI in software engineering has been for rapid prototyping. Rapid prototyping enables requirements and system designs to be iteratively refined with customers before production-quality software is manually coded. However, just as manual typing makes it difficult to modify and reuse publications, manual coding of production-quality software makes it difficult to modify, maintain, and reuse software.

In the next century, transformational programming will create a paradigm shift in software development like that created by desktop publishing. Software development and maintenance will be elevated to the specification level by automating the derivation of efficient code from specifications. Knowledge-based tools for requirement elicitation, specification evolution, and program synthesis will enable end users to specify and modify their own software. Software architectures will enable small teams of system developers to create large software systems. Advances in knowledge representation, knowledge acquisition,

and automated reasoning will enable domain experts and design experts to encode their knowledge in software architectures so that it can be reused by system developers and end users. Software engineering will be elevated to the engineering discipline of capturing and automating currently undocumented domain and design knowledge.

The path to this new paradigm will be incremental. In the coming decade, KBSE will begin to supplant CASE with more powerful knowledge-based tools. This book documents industrial pilot projects in software maintenance and special-purpose program synthesis. Current AI technology can greatly enhance conventional maintenance tools to help maintenance programmers understand a software system and determine the ramifications of changes. Current AI technology can also help reengineer existing systems. Special-purpose program synthesis systems will likely become the next generation of application generators. Compared to the technology used in existing application generators, transformational technology can produce more optimal code and provide a higher-level user interface. In short, KBSE will revolutionize the practice of software engineering by adapting to and improving current practice.

Acknowledgments

The author thanks the following people for helpful reviews: Lee Blaine, Allen Goldberg, Cordell Green, Laura Jones, Richard Jullig, David Lowry, Larry Markosian, and Stephen Westfold. The ideas in this conclusion were stimulated by the chapters in *Automating Software Design* as well as sources in the literature, numerous conversations with other members of the KBSE community, and various research initiatives sponsored by the U.S. government. The views expressed in this conclusion are the sole responsibility of the author.

References

- Abelson, H.; Eisenberg, M.; Halfant, M.; Katzenelson, J.; Sacks, E.; Sussman, G. J.; Wisdom, J.; and Yip, K. 1989. Intelligence in Scientific Computing. *Communications of the ACM* 32(5): 546–562.
- Anderson, J. S., and Fickas, S. 1989. A Proposed Perspective Shift: Viewing Specification Design as a Planning Problem. Presented at the Fifth International Workshop on Software Specification and Design, May, Pittsburgh.
- Arango, G. 1988. Domain Engineering for Software Reuse. Ph.D. thesis, Dept. of Information and Computer Science, Univ. of California at Irvine.
- Barr, A. 1990. The Evolution of Expert Systems. *Heuristics* 3(2): 54–59.
- Barstow, D. R.; Shrobe, H. E.; and Sandewall, E.,

- eds. 1984. *Interactive Programming Environments*. New York: McGraw-Hill.
- Bergland, G. D., and Gordon, R. D., eds. 1981. *Tutorial: Software Design Strategies*. Washington, D.C.: IEEE Computer Society.
- Biggerstaff, T. J.; Hoskins, J.; and Webster, D. 1989. Design Recovery for Reuse and Maintenance, Technical Report STP-378-88, MCC Corp., Austin, Texas.
- Bordiga, A.; Greenspan, S.; and Mylopoulos, J. 1986. Knowledge Representation as the Basis for Requirements Specifications. In *Readings in Artificial Intelligence and Software Engineering*, eds. C. Rich and R. C. Waters, 561–569. San Mateo, Calif.: Morgan Kaufmann.
- Brown, D. W.; Carson, C. D.; Montgomery, W. A.; and Zislis, P. M. 1988. Software Specification and Prototyping Technologies. *AT&T Technical Journal* 67(4): 46–58.
- Burson, S.; Kotik, G. B.; and Markosian, L. Z. 1990. A Program Transformation Approach to Automating Software Reengineering. In Proceedings of the Fourteenth International Computer Software and Applications Conference, 314–322. Washington, D.C.: IEEE Computer Society.
- Cleaveland, J. C., and Kintala, C. M. R. 1988. Tools for Building Application Generators. *AT&T Technical Journal* 67(4): 46–58.
- Chikofsky, E. J. 1989. *Computer-Aided Software Engineering (CASE)*. Washington, D.C.: IEEE Computer Society.
- Curtis, B.; Krasner, H.; and Iscoe, N. 1988. A Field Study of the Software Design Process for Large Systems. *Communications of the ACM* 31:1268–1287.
- Dahl, O. J.; Dijkstra, E. W.; and Hoare, C. A. R. 1972. Structured Programming. In *A.P.I.C. Studies in Data Processing*, no. 8, eds. F. Duncan and M. J. R. Shave. London: Academic.
- Despeyroux, J. 1986. Proof of Translation of Natural Semantics. In Proceedings of the Symposium on Logic in Computer Science. Washington, D.C.: IEEE Computer Society.
- Fickas, S. 1985. Automating the Transformational Development of Software. *IEEE Transactions on Software Engineering* SE-11(11): 1268–1277.
- Freeman, P., and Wasserman, A.I., eds. 1983. *Tutorial on Software Design Techniques*, 4th ed. Washington, D.C.: IEEE Computer Society.
- Gane, C. 1990. *Computer-Aided Software Engineering: The Methodologies, the Products, and the Future*. Englewood Cliffs, N.J.: Prentice Hall.
- Green, C.; Luckham, D.; Balzer, R.; Cheatham, T.; and Rich, C. 1986. Report on a Knowledge-Based Software Assistant. In *Readings in Artificial Intelligence and Software Engineering*, eds. C. Rich and R. C. Waters, 377–428. San Mateo, Calif.: Morgan Kaufmann.
- Hartman, J. 1989. Automatic Control Understanding for Natural Programs. Ph.D. thesis, Dept. of Computer Sciences, Univ. of Texas at Austin.
- Hunt, W. A. 1989. Microprocessor Design Verification. *Journal of Automated Reasoning* 5(4): 429–460.
- Jones, C. B. 1986. *Systematic Software Development Using VDM*. Englewood Cliffs, N.J.: Prentice Hall.
- Letovsky, S. 1988. Plan Analysis of Programs. Ph.D. thesis, Computer Science Dept., Yale Univ.
- Marcus, S., ed. 1989. *Machine Learning* (Special Issue on Knowledge Acquisition) 4(3–4).
- Moore, J. S. 1989. A Mechanically Verified Language Implementation. *Journal of Automated Reasoning* 5(4): 461–492.
- Mostow, J. 1989. Design by Derivational Analogy: Issues in the Automated Replay of Design Plans. *Artificial Intelligence* 40(1–3): 119–184.
- Neches, R.; Swartout, W. R.; and Moore, J. D. 1985. Enhanced Maintenance and Explanation of Expert Systems through Explicit Models of Their Development. *IEEE Transactions on Software Engineering* SE-11(11): 1337–1351.
- Rich, C., and Waters, R. 1990. *The Programmer's Apprentice*. New York: Association of Computing Machinery.
- Roth, S. F.; Mattis, J. A.; and Mesnard, X. A. 1990. Graphics and Natural Language as Components of Automatic Explanation. In *Architectures for Intelligent Interfaces: Elements and Prototypes*, eds. J. Sullivan and S. Tyler. Reading, Mass.: Addison-Wesley.
- Reubenstein, H. B., and Waters, R. C. 1989. The Requirements Apprentice: An Initial Scenario. Presented at the Fifth International Workshop on Software Specification and Design, May, Pittsburgh.
- Schindler, M. 1990. *Computer-Aided Software Design*. New York: Wiley.
- Shavlik, J. 1990. Acquiring Recursive and Iterative Concepts with Explanation-Based Learning. *Machine Learning* 5(1): 39–70.
- Shiel, B. 1986. Power Tools for Programmers. In *Readings in Artificial Intelligence and Software Engineering*, eds. C. Rich and R. C. Waters, 573–580. San Mateo, Calif.: Morgan Kaufmann.
- Spivey, J. M. 1989. *The Z Notation: A Reference Manual*. New York: Prentice Hall.
- Steier, D. M., and Anderson, A. P. 1989. *Algorithm Synthesis: A Comparative Study*. New York: Springer-Verlag.
- Warren, D. H. D. 1977. Implementing Prolog—Compiling Predicate Logic Programs, volumes 1 and 2, D.A.I. Research Reports, 39 and 40, University of Edinburgh.
- Westfold, S., and Green, C. 1991. A Theory of Automated Design of Visual Information Presentations. Technical Report KES.U.91.1, Kestrel Institute, Palo Alto, California.
- Wills, L. M. 1989. Determining the Limits of Automated Program Recognition, Working Paper, 321, AI Lab., Massachusetts Institute of Technology.
- Wing, J. M. 1990. A Specifier's Introduction to Formal Methods. *IEEE Computer* 23(9): 8–26.
- Young, W. D. 1989. A Mechanically Verified Code Generator. *Journal of Automated Reasoning* 5(4): 493–518.

Michael Lowry is affiliated with the AI Research Branch, NASA Ames Research Center in Moffett Field, California.