AUTOMATING SOFTWARE DESIGN*

Peter Freeman

NOTE: This is a preprint of a paper to be presented at the 10th Annual Design Automation Workshop.

*This work supported by National Science Foundation Grant GJ-36414

TECHNICAL REPORT NO. 30, APRIL 1973

To anyone that has had the slightest connection with the design and construction of large programs for digital computers, it is obvious that it is not a trivial task. Many times the design of such programs is woefully inadequate, and the effort to get them right is tremendous. The sometimes ruinous cost of such inadequate design is well-known.

What may not be so obvious is the almost complete lack of automation that has been applied to the task of software design. While computer scientists and technologists have been busy helping to apply computers to many other complex tasks, they have been notably tardy in applying the symbol-manipulating and decision-making power of the computer to the creation of software.

While this has been the case in the past, there has recently been an upswing in activity aimed at automating software design. The call for papers of this workshop is indicative of the heightened awareness of the problem. The purpose of this short paper is to provide some context, background, and initial roadmaps to people working in other areas of design automation who may wish to help us with the task of automating software design.

CONTEXT

As you will quickly discover, if you don't already

know, there is wide disparity in the use of many terms relating to software and programming, to say nothing of disparity in the goals that are sought. Thus, a few words about terminology are in order.

When we talk about a program we mean a set of instructions for a computer whose execution will carry out some specified task. We include in the definition, of course, a set of statements in some higher-level programming language when there exists a translator from that language to some directly executable machine language. There are shades of difference, exceptions, etc., already, but for our purposes here it will not pay to be too precise.

Software, then, can be taken to be individual programs (that perform one or a small number of functions), systems of programs that perform some more or less coordinated set of functions (e.g., an operating system), or collections of programs that perform related functions but are not intended to be used together (e.g., a library of mathematical routines). In general, we take the widest possible meaning of the word "software".

The term design is a bit more troublesome. As long as we take design to mean "the process of devising artifacts to attain goals" there is no problem. However, here we are considering software design in the context of applying

techniques and knowledge developed in other design activities. In most of the traditional areas of design automation, the end result of the process is not the final artifact (a computer, a building, a neighborhood) but rather a symbolic representation of it.

Most software designers also produce something that is not the final product. However, the symbolic representation they use is nowhere as near standardized, or complete, as an architectural floorplan, for example. Further, since software is completely symbolic, it is easy to envision representations (plans or designs) of software which are indeed software (or can become so under well-understood and automatic translations).

Thus, when talking about the automation of software design, we are in fact talking about the automation of software creation -- that is, the design, production, testing, and redesign of software using the traditional meanings of those terms. Because of the totally symbolic nature of software, however, it may turn out that techniques applicable to the design of other objects will apply to the entire creation process of software.

TWO PARADIGMS

As a means of describing the processes we want to automate, let us consider two hypothetical situations.

Semi-automated Software Creation

Suppose we have a facility primarily aimed at augmenting the human software designer. Call it TSAP, for The Semi-Automated Programmer. The human designer is able to sit down at a sophisticated terminal hooked to a powerful computing system and converse with TSAP as the first level of specification is provided. TSAP will aid and guide the designer in progressing from first rough sketches of the system to more detailed levels of design. When the point of writing and debugging programs is reached, that can be done from the terminal, as well as integrating the pieces with those built by others. Finally, as the product is tested and polished, TSAP provides support. Throughout this process, the design-aids have helped the designer communicate with other designers (current and past) as well as with programmers implementing other pieces (if it is a large system).

Without providing a complete analysis of the tasks performed (something that needs to be done), let us list a few of the things the creator does while using TSAP:

- writes functional specifications at several levels of detail;
- breaks up the task into subpieces so that other people may work on it;
- designs the structure of individual programs (again at several levels of detail);

- programs and debugs individual programs;

- integrates pieces into larger assemblages;

- tests subassemblies;

- alters specifications and pieces of the software on the basis of tests;

- communicates with other people working on the project;

- checks standards and conventions adopted for the project;

- reviews algorithms and overall designs used elsewhere.

Again without much description, let us list some of the design-aids that should be present in TSAP:

- good program preparation tools of all sorts (compilers, interpreters, editors, system features, program librarians, etc.);

- a specification language (or languages) usable at several levels of detail and able to be handled mechanically;

- a system to check design specifications against adopted standards, work in progress by other members of the design team, previous specifications at other levels of detail, and designs done on other projects;

- a "conceptual program editor" capable of selecting existing programs and/or designs and changing them slightly to meet stated specifications (i.e., the tools necessary to take "software components" and turn them into the particular item needed at the moment);

- automated testing facilities for individual programs and subassemblies, including automated resource usage measurements;

- an automated manager that does most of the things a human software project manager is supposed to do.

Pieces of this kind of design-aid system for software designers exist and others are being worked on. Large and important pieces, however, are not even well understood in terms of the functions they should provide.

Autmoated Software Creation

Now suppose we have a system aimed at removing the human from all but the functional specification phase of software creation. Let's call the program TAP (The Automated Programmer). It takes as input high-level functional and behavioral specifications of some needed software. It produces as output (a) machine code for the software and (b) documentation for the program. The software to be created may range in complexity from a sine routine to an operating system. The input may be incomplete, forcing TAP to ask the designer for more specifications, but in any event need not be stated in a highly formal or rigorous language.

The output software will be at least as good as could have been produced by other means; in most cases, it will be better. TAP will often be able to create systems whose size and complexity would make them unobtainable by other means. And TAP will create programs on a cost-competitive basis.

What would a system like TAP consist of? There is clearly a great amount of research to be done before a TAP is possible. A few of the major areas to be investigated are:

- requirements specification (especially how to handle incomplete specifications);
- general knowledge base (information every programmer carries in his head);
- program synthesis techniques (once precise specifications are developed);
- intermediate solutions (partial designs and incomplete programs);
- evaluation methods (how do we decide if the right program has been created);
- implementation methods (at what point is machine code produced);
- documentation (how to produce it automatically).

The generality and power of TAP guarantees that it won't be built tomorrow, if ever. If you are familiar with current research in artificial intelligence (ai), you will recognize here most of the unsolved problems in that area plus some more!

RANGE OF GOALS

The two paradigms given above represent only two possibilities for automating software creation. They were presented simply to provide some context for thinking about the problem. There are other goals, of course; for example,

design aids not so powerful as TSAP are already in use or being developed and a man-machine system along the lines of TAP, but again much less complete in terms of automation, could be developed. So, as in most fields, there is a continuum of goals for automation work.

As you will quickly discover, very little work has been done on software automation and it is scattered across the entire spectrum. Some of the work being called "software engineering" is relevant, as is much of the work in ai. Since we are dealing here with the central phenomenon of computer science -- programs and programming -- most research in computer science will have some eventual applicability to the task of automating software creation.

There is not much structure on the field as of yet and, as you would expect, little agreement as to what should be done. However, a growing number of people agree that something must be done. In the remainder of this paper we will provide an entry into the literature in the field and point out some areas of potential overlap with design automation in other fields.

A SAMPLING OF WORK PAST AND PRESENT

As noted above, the work on programming automation is basically fragmentary. This is not intended to be a comprehensive review of the field, so we will simply point

out a few items which will give a flavor of the work done so far.

One of the very few pieces of work before 1967 or 1968 on what we now call automatic programming is H. A. Simon's Heuristic Compiler [17]. This was an attempt to formulate programming tasks as problems that could be solved by the General Problem Solver (GPS). Although the work was very simple and restricted in scope, the paper treats the problem with a generality that makes it relevant today.

A more recent book by Simon, The Sciences of the Artificial [18], is also pertinent and should be read by anyone interested in design.

You will discover a number of papers around 1960 using the term "automatic programming" and a series of books with this title still published [1]. The term as used then referred primarily to the use of higher-level languages and was, indeed, the automation goal of that period. The term as used today, while meaning the same thing -- automating some of what programmers do -- has an entirely different context so that those earlier papers are of little interest to one interested in software design automation in 1973. Although "automatic programming" is being used by some as the generic term for all work on programming automation, it seems more logical to reserve it to describe fully automated

program synthesis.

There are some aids for the design and production of software in use by large manufacturers and others who must generate large quantities of software. Some of these are quite interesting, but generally are not openly described. The basic idea is to aid with the bookkeeping and managerial aspects of software development. Typically, they provide tools for tracking the specifications and current status of program modules, provide communication for design changes and problems, and may do a limited amount of checking for conformity to system standards. The proceedings of two NATO conferences on software engineering [19, 20] provide an entry into some of the ideas and work.

Another area with strong implications, especially for a TSAP-type design-aid, is that of structured programming. Dijkstra [4] is the chief proponent of this technique and its use may have a strong impact on programming and programming tools in the next few years (see [11] for some implications in a particular area). The rubric of structured programming includes several ideas: hierarchical structuring of a system, use of language constructs that lead to more understandable programs, and "proof" of correctness for all parts of a system before they are programmed.

Somewhat along the same lines is the work dealing with multi-level specifications [16, 23]. This work has the interesting twist of using simulations that eventually become the software being built.

The ISDOS project at the University of Michigan [21] has an ambitious set of goals aimed squarely at the automation of software design. They want to automate the creation of a significant class of business information processing systems. Input would be statements concerning the hardware configuration available, the information flow for the system (inputs and outputs), and optimization criteria. The output would be the software system to do the required task. Although it is work in progress, their results to date have not been widely disseminated.

A more recent project is the laboratory for automatic programming being developed by Cheatham and Wegbreit at Harvard [3]. Essentially, they want to provide a facility that will aid in the optimization of already existing programs. It will include measurement and simulation tools, extensible language features, and sophisticated operations to aid the programmer in transforming his program into a more optimal version. This type of project, attempting to build a practical tool with what is now available, will probably become much more prevalent in the next few years.

There are several groups aiming at this level of aspiration.

There are two, not un-related, lines of work that have great theoretical interest for software automation and may eventually prove of practical importance. The work on program verification stems largely from the early work of Floyd [9]. Given a program, one forms an interpretation of it by attaching verification statements to all points in its flow-chart. The statements normally make some assertion about the state of the machine at that point in the execution of the program. If these statements can be shown to follow logically from each other then an interpretation of what the program does will have been verified. Unfortunately, human error can creep into the process of drawing the interpretation or the proof of verification statements and the power of mechanical theorem provers severely limits automation. Much of this work is surveyed in [5]. About 1968, Green [12] and Waldinger [22] independently demonstrated the possibility of generating programs from proofs in the first-order predicate calculus. The program to be written is stated in the form of a theorem and the operations of the language are stated as axioms. If the theorem can be proved, then their techniques permit one to construct the desired program from the steps of the proof. A good bit of work has flowed from this start (see

[14] for a review) but is limited in effectiveness by the power of the mechanical theorem provers currently available.

There is other work of theoretical interest to software design automation ranging from problem-solving techniques in ai [8] to models of design [10] to mathematical results on the properties of programs [6]. As more coordinated projects develop, some of these results may come into play.

There are almost no published surveys of the state-of-the-art in software automation (a comment in itself). The Cheatham and Wegbreit paper [3] has a brief survey and two technical reports [2, 7], especially the one by Balzer, cover some of the relevant literature.

SOME AREAS NEEDING WORK

Whether one is concerned with complete or partial automation of software creation, there are several ways of viewing the problem. The view one adopts or develops will govern the relative importance of the following areas. We present them here, however, without interpretation in the hope they will suggest fruitful applications of existing techniques.

Problem Representation

In specifying that a certain program is to be written, the nub of the problem is to give sufficient information

without actually writing the program. In some cases a process description is the most efficient way of stating the problem and in others a state description is best. Finding suitable ways of uniformly stating programming problems will be an extremely important part of any software design automation.

When one considers the problem of designing a large system of programs, there is the added problem of representing the desired interaction between separate functional units. Given a way of specifying that large system, the problem is not whipped, however. Now, we must be able to proceed from a high-level description to a more detailed problem statement in which design specifications for individual programs are given. This is the same problem described above, except that here we are asking that the problem representations be generated (potentially) automatically.

A critical piece of the problem representation concern is the determination of what is missing from a given statement. The techniques for doing that, for obtaining the additional problem information, and for refining the problem statement may well be generalizable from other areas of design automation.

Solution Representation

The final solution, a program, is a symbolic object and requires no representation other than itself. However, there are representational problems we must deal with. In developing a solution, we may have to deal with several levels of a design in terms of increasing specificity. These intermediate levels may be considered solutions in that they indicate structures and their interconnections. Of a similar nature are partial solutions that are incomplete in one or more ways. They may not meet some criteria or constraints, they may have some piece of the solution missing, or they may emphasize only one aspect of the solution.

We need to represent solutions in order to evaluate them. As with any design, there are usually several programs (often very many) that will meet our existence criteria. The problem is to pick the best with respect to stated criteria and to do this we must be able to manipulate and evaluate a solution easily. For most such processes, a set of machine instructions will not be the best representation (if for no other reason, because it will be too voluminous).

The problem of program documentation is essentially finding a representation of a program that is easily understandable to humans. It is not unlike the problem of

abstracting an article. The documentation problem is similar to finding good ways of representing partial and intermediate solutions, but has some different aspects.

Representation of General Knowledge

One of the areas in which software automation faces the same problem as many other efforts is in capturing and encoding a sufficient set of general knowledge about the problem domain. Software designers, like others, carry a vast amount of useful information in their heads: characteristics of the physical domain in which they work, partial solutions to standard problems, problem-solving methods, etc.

First, it is necessary to determine which of this information is needed for an automated (or even semi-automated) software designer. The problem is especially prevalent for software automation because so little work has been done on studying the programming process or the characteristics of programs. (For instance, the notion of a software component is not well-defined and there is certainly no handbook giving the charcteristics of standard pieces of software.)

Once we have determined what general knowledge is necessary along with what domain-specific knowledge is mot useful, we still have the problem of representing it in a

machineable form that will permit its easy use. Hopefully, the similarity of this problem with other design automation efforts will provide some useful clues.

Problem-Solving

No matter the level of automation we seek, the more "intelligence" we can build into our machine system the better. This comes down to the power of the problem-solving methods available and in that we are no different from other design automation areas. We both must draw from ai research.

However, design automation does have a particular set of needs. We need automated problem-solving tools that are quite powerful in limited domains. We must figure out ways of using domain-specific information as aids to more general problem-solving methods (such as heuristic search or resolution theorem proving). To draw from ai research, we must in turn contribute to it by helping develop the techniques of using the domain-specific information we have. Our knowledge of how humans use this information in design will perhaps give an important clue.

Global Optimization

Large software systems, like many systems, face a difficult optimization problem that is one of the key

stumbling blocks to design automation. While it is often clear how to perform local optimization it is not clear at all how to optimize globally in a system where uniform local optimization may indeed pessimize overall performance.

With software, the basic tradeoff is between time and space. Simply being able to measure values realistically before the system is complete and being executed is the first problem. Being able to use the information to change the design for the better is the next step. Similar problems of global optimization in other areas may provide some insight.

SUMMARY

After providing encouragement to tackle a new set of problems, a word of caution is in order. Many design problems, as you are no doubt aware, appear similar, if not identical, at a high-enough level. Yet, it may be the domain-specific information that provides the key. Unless software design problems can be formulated in detail exactly like some other class of design problems, the use of techniques from other areas may require a good deal of work. Further, the problems of representation mentioned above are not new to computer science. There is no question that they are tough problems.

Programming automation is long overdue and there are a

number of current efforts to correct this deficiency. The range of goals being sought is large, running from slightly more intelligent compilers than we now have to completely automated programmers. The amount of work to date on software automation is pitifully small, however.

Our aim has been to introduce researchers interested in other areas of design automation to the problems that will be encountered in software automation. Hopefully, brief descriptions of some of the areas where design automation techniques are needed will spur you into digging into them more deeply. If you do, perhaps you will discover that what you have used successfully in your field will work with software design automation.

REFERENCES

1. Annual Review of Automatic Programming, Academic Press.

2. Balzer, R.M., "Automatic Programming," Item 1, University of Southern California Information Sciences Institute, September 1972.

3. Cheatham, Jr., T.E., and B. Wegbreit, "A Laboratory for the Study of Automating Programming," 1972 Spring Joint Computer Conference.

4. Dahl, O., E. W. Dijkstra, and C. A. R. Hoare, Structured Programming, Academic Press 1973.

5. Elspas, B., K. N. Levitt, R. J. Waldinger, and A. Waksman, "An Assessment of Techniques for Proving Program Correctness," ACM Computing Surveys, June 1972.

6. Feldman, J. A., "Some Decidability Results on Grammaticalinference and Complexity," Information and Control, 1972.

7. Feldman, J. A., "Automatic Programming," Stanford Artificial Intelligence Project Memo Aim-160, Stanford University, February 1972.

8. Fikes, R. E., and N. Nilsson, "STRIPS, A New Approach to the Application of Theorem Proving to Problem Solving," Proceedings Second IJCAI, London, 1971.

9. Floyd, R. W., "Assigning Meanings to Programs," Proceedings of Symposia in Applied Mathematics, American Mathematical Society, Vol. 19 (1967).

10. Freeman, P., and A. Newell, "A Model for Functional Reasoning in Design," Proceedings Second IJCAI, London, 1971.

11. Freeman, P., "Functional Programming, Testing and Machine Aids," Program Test Methods, Prentice-Hall, 1973.

12. Green, C. C., "Application of Theorem Proving to Problem Solving," Proceedings First IJCAI, Washington, D.C., 1969.

13. King, J. C., A Program Verifier, Carnegie-Mellon University, Ph.D. Thesis, 1969.

14. Monna, Z., and R. J. Waldinger, "Toward Automatic Program Synthesis," Communications of the ACM, March 1971.

15. Mills, H., "Top-down Programming in Large Systems," Debugging Techniques in Large Systems, Prentice-Hall, 1971.

16. Parnas, D.L., and J.A. Darringer, "SODAS and a Methodology for System Design," 1967 Fall Joint Computer Conference Proceedings, Thompson Books, Washington, D.C.

17. Simon, H.A., "Experiments with a Heuristic Compiler", Journal of the Association for Computing Machinery, October 1963.

18. Simon, H.A., The Sciences of the Artificial, MIT Press, Inc., 1969.

19. Software Engineering, Editors: P. Naur and B. Randell, Report on a conference sponsored by the NATO SCIENCE COMMITTEE, Garmisch, Germany, 7th to 11th October, 1968.

20. Software Engineering Techniques, Editors: J.N. Buxton and B. Randess, Report on a conference sponsored by the NATO SCIENCE COMMITTEE, Rome, Italy, 27th to 31st October, 1969. (Available from the Scientific Affairs Division, NATO, Brussels, Belgium.)

21. Teichroew, D, and H. Sayani, "Automation of System Building," Datamation, August 15, 1971.

22. Waldinger, R.J., and R.C.T. Lee, "PROW: A Step Toward Automatic Program Writing," Proceedings First IJCAI, Washington, D.C., May 7-9, 1969.

23. Zurcher, F.W., and B. Randell, "Iterative Multi-level Modeling -- a Methodology for Computer System Design," Proceedings of the IFIP Congress 1968, North-Holland Publishing Co., Amsterdam, 1968.