Portability by Automatic Translation A Large-Scale Case Study

Yishai A. Feldman and Doron A. Friedman
Dept. of Computer Science
Tel Aviv University
69978 Tel Aviv, Israel
E-mail: {yishai,doronf}@math.tau.ac.il

Abstract

Automatic code translation could be a useful technique for software migration, provided it can be done in large-scale industrial applications. We have built an automatic translation system for converting IBM 370 assembly-language programs to C, in order to port the original programs to different architectures. This system, called Bogart, first analyzes the original program in terms of data flow and control flow, and translates it into an abstract internal representation. It performs various transformations on the abstract representation, and finally re-implements it in the target language.

Bogart was successfully tested on several large modules with thousands of lines of assembly code each, taken from a commercial database system and application generator. The results of this research are compared with the brute-force approach first implemented by the company, showing Bogart to be superior on all counts. This research is unusual in that it took place in industry, and had a clear objective of translating a real program and not just demonstrating the feasibility of the approach in an academic setting. Lessons from this experience are discussed at the end of the paper.

1 Introduction

Many organizations today are facing the problem of porting existing code to new architectures and operating systems. In many cases, such legacy code is written in a mainframe-specific assembly language and needs to be translated to a high-level language (HLL) in order to be run on different architectures. Our research focused on one such example, a large database system and application generator written in IBM 370 assembly language, the main product of Sapiens International, Ltd.

This research was supported in part by a grant from the Israel Ministry of Science and Technology.

The Sapiens application consists of several hundred thousand lines of manually-optimized code, developed over two decades. A few years ago, Sapiens decided to port their application to other architectures, most notably Unix workstations and PCs. At the same time, the main product running on IBM mainframes had to be supported and maintained. While HLL code performance was expected to be adequate for modern workstations, the same was not true for large databases running on mainframe computers, and therefore the original assembly-language code had to be retained. Because the cost of manually re-writing the code at the same time as maintaining and developing the assembly-language version was too prohibitive, and backward-compatibility was paramount, this option was ruled out. Sapiens therefore turned to automatic translation as a cheaper solution. It was important that the HLL code be generated completely automatically from the assembly-language source, in order to allow continuing modification of the latter. However, readability of the HLL code was not considered important at this stage.

In 1992 Sapiens started to develop an assembly-language-to-C translator. They used a "literal" approach in which each source line was translated to a single C statement, and an array in memory was used to simulate the registers of the IBM 370. In effect, the result of the translation was an IBM 370 simulator partially evaluated with the application program. Since it was clear that this simple-minded approach would not be sufficient, it was to be aided by extensive re-writing of the original assembly-language sources, and guidelines for this code improvement were drawn up. These included generally beneficial improvements such as the eradication of techniques that have no parallel in high-level languages and are even bad practice in assembly language, and the use of macros for structured programming constructs such as conditionals and loops. However, also included were special macros that specified C code to be inserted into the translated code directly, as well as other harmful changes from the assembly-language programming point of view.

2 The AI Solution

The situation at Sapiens was thus an excellent opportunity for testing AI approaches to the translation problem as well as for comparing it with the industrial brute-force approach. It was clear to us that the literal translation approach, besides being grossly inefficient, would also fail to be portable, because it fails to recognize idioms that are architecture-specific and would not have the same meaning when translated literally. Examples are the use of the high-order bit of a pointer as a flag (depending on the assumption that pointers are at most 31-bits long), and access to parts of multi-byte entities (depending on byte order within words).

In 1993 we started developing, together with Sapiens, a different automatic translator based on abstraction and re-implementation [1], and called Bogart.[1] Both translators were implemented and used, proving Bogart to be superior to the brute-force translator on all counts (detailed comparisons are presented below).

Bogart is based on the premise that a program is conveniently manipulated not in a textual format, but is best represented in terms of data flow and control flow. (This also seems to fit the way programmers think about their code.) The first step Bogart performs when translating a program is therefore data flow and control flow analysis, and the construction of an abstract representation, similar in spirit to the Plan Calculus [2]. Once an abstract representation of the program is constructed, it is possible to transcend the details of the source language that are irrelevant to the algorithm. For example, one of the fundamental aspects of assembly-language programming, the use of registers, is abstracted away because registers are only used for effecting the flow of data. The second stage in the translation consists of further analysis and transformations of the abstract representation. Cliché recognition and other advanced techniques could be added to this stage. The last stage is the re-implementation of the algorithm in the target language. This is a relatively simple step, mainly because the resulting code was not required to be particularly readable.

[1]Better Optimizing General-purpose Abstract Representation Translator, also named after the second author's dog.

Currently, Bogart performs mainly local analysis of its input program. While this has been sufficient to produce code that is superior and more portable than that of the simulating translator, we expect that global analysis would be required for further improvements.

Bogart's performance can be illustrated by one of the small examples for automatic translation, the Horner routine, taken from an IBM 360 assembly textbook [3]:

```
HORNER  CSECT
        STM    R14,R12,12(R13)
        LR     R12,R15
        USING *,R12
*     ** INITIALIZATION **
        LA     R7,COEF
        L      R5,0(R7)     SUM = A0
        LA     R9,0         I = 0
*     ** TEST FOR EXIT **
LOOP    CR     R9,R2
        BNL    OUT
*     ** ADJUSTMENT STEP **
        LA     R9,1(R9)
        LA     R7,4(R7)     NEXT COEFFICIENT
*     ** BODY OF LOOP **
        MR     R4,R3        SUM*X
        A      R5,0(R7)     SUM = SUM*X + AI
        B      LOOP
OUT     LR     R0,R5
        LM     R1,R12,24(R13)
        BR     R14
COEF    DS     10F
        END
```

The simulating translator generated the following code:

```
void  HORNER (tagSAPReg *Reg)
{
    T_stm(14,12,((Reg[13].ucp+12)),Reg) ;
    Reg[12].sw = Reg[15].sw ;
    Reg[7].pv = &(COEF[0]) ;
    Reg[5].sw = *(sWord *)Reg[7].ucp;
    Reg[9].sw = 0 ;
LOOP:
    if ((Reg[9].sw)>= Reg[2].sw) goto OUT;
    Reg[9].sw += 1;
    Reg[7].sw += 4;
    T_mult(&Reg[4],Reg[3].sw) ;
    Reg[5].sw += *((sWord *)(Reg[7].ucp)) ;
    goto LOOP ;
OUT:
    Reg[0].sw = Reg[5].sw
    T_lm(1,12,((Reg[13].ucp+24)),Reg) ;
    return;
}
```

As can be seen from this example, the simulating translator relies on an array of a union type to simulate the assembly registers, and literally translates each assembly instruction into a corresponding C statement. If a corresponding C operator is not available, a library function is used to translate the operation. An example is multiplication, which generates a 64-bit result on the IBM 370 but only a 32-bit result in C.

The following code was produced by Bogart for the same routine:

```
sWord HORNER(sWord r2sw, sWord r3sw)
{
    sWord    r5sw;
    sWordPtr r7swp;
    sWord    r9sw;

    r7swp = (sWord *)(&COEF[0]);
    r5sw = *r7swp;
    r9sw = 0;
    while (r9sw < r2sw) {
       r9sw++;
       r7swp++;
       r5sw = r5sw * r3sw + *r7swp;
    }
    return r5sw;
}
```

This example demonstrates several important ways in which Bogart produces shorter and more efficient code. (For each point there are better and more impressive examples, but presenting them all would require more space than available here.)

- *Removal of redundant code.* Some assembly instructions do not have to appear in the high-level code at all. For example, the first few and last few assembler instructions have to do with OS/370 calling conventions and register save-areas, which are irrelevant to other architectures and supplied by the C compiler on OS/370.

- *Combination of expressions.* Assembly language does not support compound expressions, and therefore neither does the simulating translator. Because of the data-flow analysis it performs, Bogart can collect several instructions into one C statement, even if the assembly instructions are not consecutive.

  In the example, Bogart was able to generate the single statement `r5sw = r5sw*r3sw + *r7swp;` instead of two statements generated by the simulating translator.

- *Removal of computations of unused results.* Certain machine instructions generate results that are not used by subsequent code. This can happen because the instruction is used for effect rather than for value, or because multiple results are generated. An example of the latter case is multiplication, which on the IBM 370 generates a 64-bit result, placed in a pair of 32-bit registers. Often, it is known that the actual result is only 32 bits long, and the upper part of the result is ignored by the assembler programmer.

  The simulating translator has no information about the use of such results, and is therefore required to generate them in every case. For multiplication, this means that a function from a special simulation library must be called to calculate the full 64-bit result. Bogart can identify those cases in which a result is not used, and can therefore translate the multiplication in terms of the 32-bit C operator.

  Similar effects can be seen in the case of integer division, which produces a quotient as well as a remainder, only one of which is subsequently used in many cases.

  A prime example of the same phenomenon is the setting of the condition code. As in many other hardware architectures, many IBM 370 instructions set a condition code, consisting of two bits in the Program Status Word, to indicate the results of the operation. All comparison instructions, most arithmetic instructions, and many other instructions set the condition code. While in many cases the condition code is ignored by the assembly-language programmer, it may be tested by one or more subsequent instructions, not necessarily adjacent.

  The simulating translator was enhanced with special ad-hoc code to recognize the common case in which a conditional branch instruction immediately follows a comparison instruction. This code is based on the assumption that the condition code is not tested any further at the destination of the branch; this assumption is statistically reasonable but can generate subtle bugs when violated. (Unfortunately, the simulating translator does not even issue a warning in this case.) However, the following code fragment, taken from the Sapiens module GREDCE, stumps the simulating translator, because of the logical shift (SRL) instruction that intervenes between the setting of the condition code by the compare (CH) instruction and its subsequent use by the branch instructions (BH, BNH):

```
CH      R7,0(R5,R4)
SRL     R2,1
BH      CGADD
BNH     CGSUB
```

The simulating translator generates the following code for this fragment:

```
if (Reg[7].sw ==
    SH(Reg[4].ucp+Reg[5].sw))
   __CC = _CZero;
else if (Reg[7].sw <
         SH(Reg[4].ucp+Reg[5].sw))
   __CC = _COne;
else __CC = _CTwo;
Reg[2].uw >>= 1;
if (__CC & 0x4) goto CGADD;
if (__CC & 0x3) goto CGSUB;
```

Bogart, in contrast, analyses the data-flow of the condition code and can therefore generate the following code:

```
r2sh >>= 1;
temp = SH(r4ucp + r5sh);
if (r7sh > temp) goto CGADD;
if (r7sh <= temp) goto CGSUB;
```

- *Computation of routine interfaces.* The simulating translator has no information about subroutine interfaces, and therefore passes the array corresponding to the machine's 16 general registers to every subroutine. The subroutine can change the values of some of the simulated registers, thus passing results back to the calling routine.

  Bogart analyses the usage of registers inside each routine, and can thus recover the actual interface: which registers are used for input, output, or both. It can therefore declare the subroutine with a number of parameters corresponding to the registers it actually uses, and, in the case of a single returned value, returns it as the value of the function.

- *Type analysis.* Bogart performs limited and local type analysis (future extensions are described later in this paper). This enables it to generate more portable and more readable code. In contrast, the simulating translator relies heavily on type casting, with its attendant portability risks. A small example of this can be seen in the Horner routine, where Bogart was able to deduce that register 7 contains a pointer to a 32-bit word, and could therefore declare it as such and generate the concise and portable `r7swp++` instead of the simulating translator's `Reg[7].sw += 4`.

- *Recognition of control structures.* Readability was only a secondary goal in this case, because the target code was not meant to be handled by human programmers. However, simple control structures such as if-then-else and while loops were recognized by Bogart with little effort.

3 Results

Bogart was tested on several Sapiens modules. SAPDBMS, a central Sapiens module, was chosen by Sapiens management as a major test case. A basic database transaction (such as insert, find, delete, or map) enters SAPDBMS at least a hundred times and potentially more than a thousand times. SAPDBMS was integrated into a working subsystem and compared with the version produced by the simulating translator. Bogart was also tested on several small routines taken from an IBM 360 assembly-language textbook [3]. For obvious reasons, these could be tested on more platforms and more measurements could be performed on them.

Bogart's performance was found to be superior to that of the literal translator on all counts, as detailed below. The two translators had different sets of requirements from the company and its programmers; these are discussed in the final section of the paper.

3.1 Portability

As expected, Bogart produced more portable code than the literal translator. The code translated by the literal approach is expected to run only on systems with 32-bit word and pointer sizes, flat memory model, and big-endian byte order. In contrast, Bogart produced code that was successfully executed on an AS/400 machine,[2] for which the literal translator

[2]The AS/400 pointer size is 128 bits!.

failed. Translation by abstraction also allows supporting a larger portion of the source language. As work progressed, more and more cases were found which the literal translator could not handle correctly without more global information. This led to further requirements from the programmers modifying the source, with the undesirable effects described in the next subsection.

The only method that will allow the literal translator approach to produce more portable code is by additional manual work. In contrast, the abstraction approach can be improved by deeper analysis, such as the suggested constraint-propagation component for type analysis (see Section 4).

3.2 Manual Preparation

Translation by abstraction requires less manual work, since it enables the translator to use the available global information to support larger portions of the original code. Since Bogart used the code that had already been manually processed for the literal translator, it is impossible to quantify the difference, but it is clear that several of the "improvements" necessary for the literal translator are not necessary for the abstracting translator, and may even degrade its performance. For example, literal constants in the assembly-language code were converted into variables, thus losing the important information of their immutability and forcing Bogart to translate them to C variables instead of constants.

Manual modification of Sapiens code was found to proceed at a pace of about 3600 lines of code per person-month. Since rewriting the whole system in C was estimated to require 100 person years, the preparation time was considered reasonable by management. However, it turned out to be a tiresome job with serious undesired effects on staff morale.

Manual preparation of the code has probably damaged the code's quality. Programmers estimate that the code is less efficient after standardization, and, naturally, new bugs were introduced. In order to avoid introducing errors, many programmers over-used the ability to write C code that coexists with the original assembly code. This violated one of the major requirements of the translation project—two versions now had to be debugged, tested, and maintained. This is an important lesson: extensive manual work is not only harmful for the resources it requires, it may also endanger the whole translation enterprise.

3.3 Efficiency

Bogart produced much more efficient code, in terms of both space and time. Typically, Bogart code was between half and three quarters as large and more than twice as fast as the literal translator's output. This is due to the abstraction performed by Bogart and to the optimizing transformations performed on the abstract representation. Bogart is even able to improve on parts of the original assembly-language sources. Our experience with Sapiens code has shown that large and complex assembly-language programs that are maintained by several different programmers contain patches and unnecessary code, such as loading a register with a value already present in it. Such cases are discovered by the abstraction analysis performed by Bogart, and are removed in the transformation phase of the translation.

Table 1 presents some of the results for the SAPDBMS module described earlier, as well as for three small programs (Bin, Horner, and Random, taken from an IBM 360 assembly textbook [3]). Times for SAPDBMS were computed for a sequence of thousands of transactions running in batch. All programs were run on an RS/6000 machine under AIX.

Table 2 shows more detailed comparisons of the program Bin (a binary search program) on different platforms and compilers. Shown are times for both translators, a manually hand-crafted C program, and (for the IBM 370) the original assembly-language code. (The program was run in a loop, with different numbers of iterations on different platform.)

A striking result was achieved with some small examples—the code produced by Bogart slightly outperformed code written in the target language by a human programmer! This is probably due to the fact that in these small programs using structured programming constructs is less efficient than direct jumps out of multiple-exit loops, as is natural when programming in assembly language.

More significant is the comparison between the original assembly code and the result of Bogart's translation. The translated C code was only three times slower than the original, which in our opinion is quite reasonable for such translation. In fact, it is less than 10% slower than the hand-crafted C version on the same platform. Unfortunately, we could not test the translated SAPDBMS module on the IBM mainframe, but we expect similar results.

Table 1: Results of translated programs on RS/6000

	Literal Translator Output		Bogart Output	
	Time (sec.)	Space (bytes)	Time (sec.)	Space (bytes)
Bin	63	4170	33	2802
Horner	10	3302	3	2465
Random	9	5447	4	2741
SAPDBMS	18	41700	9	29073

Table 2: Running time of BIN program on various platforms (in sec.)

	RS/6000	Microsoft C 7.0	Borland C	IBM 370	AS/400
Original (assembly)	—	—	—	1.14	—
Hand-crafted C	32	36	39	3.18	52
Literal translator	63	46	42	6.08	*failed*
Bogart	33	35	35	3.49	107

4 Future Work

Although Bogart is now capable of adequate translation, it does not implement the full theoretical framework of translation by abstraction. The current version of Bogart performs mostly local analysis, and relies on reasoning from first principles rather than on chunked knowledge. It needs to be extended to do more global analysis and possibly use a knowledge library.

The most significant improvement in Bogart's capability to generate high-quality portable code would come from the addition of a data-type recognition component. Assembly language requires very little type information, and does not provide the means to describe complex structures. For the purposes of writing assembly code, there is little difference between a 32-bit integer, a pointer, or a 4-byte string. However, the way in which a given location is accessed can provide some information about its type. For example, storing the result of an arithmetic addition operation into such a location indicates that it contains an integer or a pointer. Information about the operands of the addition operation can further restrict the possible types of the result; for example, if it is known that neither operand is a pointer, the result cannot be a pointer either. When a value is dereferenced, it is clear that it is a pointer. In this case, similar reasoning can be used to deduce the type of the object it points to.

In this way a constraint-propagation network can be constructed from the original assembly-language program. This network connects all storage variables and dereferenced pointers through the operations they participate in, and type information can be refined by reasoning of the kind shown above. We have designed such a type reasoner, but it has not yet been implemented.

The unstructured and undisciplined nature of assembly-language programming and the fact that C is a relatively low-level language dicatate a first-principles approach to the automatic translation task. Any solution that relies on the recognition of specific idioms in the assembly-language program is likely to be limited to a small part of the program. However, in certain cases the recognition of high-level clichés [4] can aid the translation effort, particularly in the analysis of data structures.

An example in which cliché recognition could be useful is the recognition of variable-length lists of pointers. Since pointers on the IBM 370 are either 24 or 31 bits long, such lists are typically represented as vectors of pointers, with the most-significant bit used to indicate the end of the list. If the clichés of building or accessing such lists can be recognized, the data structure can be replaced by a different representation, more typical to C programming, such as an additional integer storing the length of the list. However, at present it seems that the current state-of-the-art in this area is not yet applicable to large-scale programs.

5 Related Work

Abstraction of programs has been used for many purposes, including automatic documentation, modularization, recovery of reusable components, translation, and even compilation. Most research efforts have been tested on small examples, with a few noteworthy exceptions. Abstraction systems also differ in the amount of interaction they require from their users.

Bogart is most similar in spirit to Faust's SATCH [5], which translates Cobol programs to the more abstract non-procedural language HIBOL. SATCH, like Bogart, represents programs in a formalism based on the Plan Calculus [6]. SATCH performs deeper analysis of its source code, but it has not passed the stage of a demonstration system, and has only been tested on a few small examples.

Another translation effort was the transformation-based conversion of the program transformation system TAMPR from a highly abstract form (pure applicative LISP with dynamic scope) to Fortran [7]. 1300 lines of LISP were translated into 3000 lines of Fortran, with a 25% gain in performance. However, This is not a typical example. Because of the unusually abstract way in which the source program was written, all that was necessary was to add design decisions in order to generate a concrete program in Fortran. In contrast, Bogart's main effort was abstracting away from the details of the source program. For example, the TAMPR translator had to choose a specific order of execution where LISP leaves it unspecified, whereas Bogart relaxes the overspecified order of the assembly-language program.

Cobol/SRE [8] and Refine/Cobol [9] use abstraction for recovery of reusable components from and modularization of legacy Cobol programs. Both are interactive tools allowing the user to manipulate the source program in various ways. Cobol/SRE has only been tested in the laboratory, but Refine/Cobol has been used to re-engineer modules of up to 40,000 lines of code.

As mentioned above, Bogart does not yet perform recognition of high-level clichés. While such a capability might be useful, research on this problem [4, 10, 11, 12] has been confined to small examples, and no way to control the inherent combinatorial explosion of the recognition task has yet been found.

6 Discussion

This work can be considered as an "industry as laboratory" case study in the spirit advocated by Colin Potts [13]. It was an excellent opportunity to compare the AI approach with a typical industry approach. We believe our research showed that the AI approach can be more successful and cost-effective than the "brute force" approach, but its limitations were also made clear.

The main advantage of the literal translation approach is its simplicity. The core of the literal translator was implemented in three months by one programmer. The corresponding part of Bogart required 30 person-months, and relied on existing framework (such as parser and data dictionary) from the literal translator.

Another surprising advantage of the literal translator was found in the Sapiens case study. After translation, the resulting code has to be tested and debugged. Debugging is done by the assembly-language programmers, who are familiar with the original code. They found it much more convenient to work with the literal translation: after getting used to the C syntax, they could recognize the original code in the translated code, and could use their existing debugging methods and knowledge of the code. Debugging Bogart-generated code was much more difficult for them, since they found it much harder to recognize the original code. This problem could of course be alleviated by an intelligent browsing tool for Bogart that could explain the assembly origins of the resulting C code. Currently, this information is only available in the form of comments in the translated code.

The AI approach often requires a large effort in preparing a foundation. The case described here may be representative, in that the efforts pay off only for the long term, and only if enough resources are allocated. Bogart now is about to produce a Sapiens version that is twice as efficient as that produced by the literal translator. If it had been finished earlier, it would have saved a significant amount of manual work. Since versions of Sapiens are needed on other platforms, the benefit is expected to grow. With further development, it is also expected to aid in code maintenance and debugging.

None of this could be said about the literal translator; however, it was crucial as a short-term solution. Moreover, the changes in the external and internal conditions were too rapid for Bogart's development pace; although it has proved its utility, it is not clear at the moment to what extent it will be utilized by the organization. Thus, the attempt to use AI technology should take into account the relatively long development times and heavy investment required. The development time could have been reduced, but companies hesitate to invest large resources on experimental projects.

Some conclusions about academic and industrial cooperation can also be drawn from this research. It is important to note that the organization had little interest in automatic translation per se, and was only interested in the best translation possible in terms of the target quality and investment of resources. The cooperation was convenient for both sides, and we had full access to the data, services from the literal translator, and full cooperation from company staff. Organizational considerations dictated some parts of the work, but on the whole we were free to choose the architecture of the translator. We therefore consider this case to be a successful cooperation, but unfortunately such opportunities are quite rare.

A large part of the development of Bogart was dedicated to many theoretically-unimportant details, such as supporting most of the IBM 370 instruction set. This is of course crucial to the industrial translation effort, and is unavoidable if we want to prove that a theoretically-elegant approach is applicable to the real world. However, it is impossible in a purely academic setting, and requires support and help from industry.

An important aspect of the abstraction approach to translation is its generality. We believe that the same approach could be used to translate other assembly languages. Although other architectures contain different idiosyncrasies that might require special treatment, the general framework of abstraction, transformation, and re-implementation should still be applicable. In addition, the information collected during translation could be used to support other reengineering tasks.

Acknowledgments

We are grateful to Sapiens International Ltd. for their support of this project, and in particular to N. Barzilay, A. Cohen, I. Judkevitz, Y. Kazmirsky, and E. Kiril. A. Yehudai provided useful comments on this paper.

References

[1] R. C. Waters, "Program translation via abstraction and reimplementation," *IEEE Trans. Software Engineering*, vol. 14, pp. 1207–1228, Aug. 1988.

[2] C. Rich and R. C. Waters, *The Programmer's Apprentice.* ACM Press and Addison Wesley, 1990.

[3] G. Struble, *Assembler Language Programming: The IBM System/360.* Addison Wesley, 1969.

[4] L. M. Wills, "Automated program recognition by graph parsing," Technical Report 1358, MIT Artificial Intelligence Lab., July 1992. PhD thesis.

[5] G. Faust, "Semiautomatic translation of COBOL into HIBOL," Technical Report 256, MIT Lab. for Computer Science, Mar. 1981. Master's thesis.

[6] C. Rich, "A formal representation for plans in the Programmer's Apprentice," in *Proc. 7th Int. Joint Conf. Artificial Intelligence*, (Vancouver, British Columbia, Canada), pp. 1044–1052, Aug. 1981. Reprinted in M. Brodie, J. Mylopoulos, and J. Schmidt, editors, *On Conceptual Modelling*, pages 239–270, Springer-Verlag, New York, NY, 1984, and in C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[7] J. M. Boyle and M. N. Muralidharan, "Program reusability through program transformation," *IEEE Trans. Software Engineering*, vol. SE-10, pp. 574–588, Sept. 1984.

[8] J. Q. Ning, A. Engberts, and W. Kozaczynski, "Automated support for legacy code understanding," *Comm. ACM*, vol. 37, pp. 50–57, May 1994.

[9] L. Markosian, P. Newcomb, R. Brand, S. Burson, and T. Kitzmiller, "Using an enabling technology to reengineer legacy systems," *Comm. ACM*, vol. 37, pp. 58–70, May 1994.

[10] M. T. Harandi and J. Q. Ning, "Knowledge-based program analysis," *IEEE Software*, vol. 7, pp. 74–81, Jan. 1990.

[11] A. Quilici, "A memory-based approach to recognizing programming plans," *Comm. ACM*, vol. 37, pp. 84–93, May 1994.

[12] P. A. Hausler, M. G. Pleszkoch, R. C. Linger, and A. R. Hevner, "Using function abstraction to understand program behavior," *IEEE Software*, vol. 7, pp. 55–63, Jan. 1990.

[13] C. Potts, "Software-engineering research revisited," *IEEE Software*, vol. 10, pp. 19–28, Sept. 1993.