ELSEVIER

Artificial Intelligence 107 (1999) 1–28

Artificial Intelligence

Portability by automatic translation: A large-scale case study ☆

Yishai A. Feldman [a,*], Doron A. Friedman [b,1]

[a] *School of Computer and Media Sciences, The Interdisciplinary Center, P.O. Box 167, 46150 Herzliya, Israel*
[b] *Department of Computer Science, Tel Aviv University, 69978 Tel Aviv, Israel*

Received 16 August 1996

Abstract

Many organizations today are facing the problem of *software migration:* 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 in order to be run on different architectures. Our research addresses this problem in a large-scale, real-life case study. We built an automatic tool, called Bogart, that translates IBM 370 assembly language programs to C. Bogart is based on Artificial Intelligence tools and techniques such as the Plan Calculus, translation by abstraction and re-implementation, program transformations, constraint propagation, and pattern recognition.

Bogart was tested on real legacy code of a large commercial application: a database system and application generator, the main product of Sapiens International, Ltd. Bogart is compared with the literal brute-force translator initially developed by Sapiens, and is found to be superior on all counts, including portability of the resulting code, the amount of manual preparation required, and code size and speed. The results are shown for several small examples as well as a typical module consisting of several thousand lines of code from the Sapiens application. Bogart also seems to be more comprehensive than other reengineering systems reported in the literature. Bogart's analysis technology has recently been applied with significant commercial success to the analysis and remediation of Year 2000 bugs.

This study demonstrates that certain AI techniques can be carefully combined to create industrial-strength applications that solve acute problems of Software Engineering. The fact that the research was carried out in industry on a real test case also revealed some of the problems of this approach. One example is the higher development cost of the AI approach, and the further effort that will be

☆ This is a revised and expanded version of a paper presented at the Tenth Knowledge-Based Software Engineering Conference, Boston, November 1995. This research was supported in part by a grant from the Israel Ministry of Science and Technology.

* Corresponding author. Email: yishai@idc.ac.il. http://www.idc.ac.il/yishai.

[1] Email: doronf@math.tau.ac.il. http://www.math.tau.ac.il/~doronf.

PII: S0004-3702(98)00101-5

needed in order to extend it. (On the other hand, the literal translator has reached the end of its road, and cannot be enhanced at all.) Another problem we discovered is the difficulty of debugging the code produced by Bogart. The literal translator preserved the structure of the original program, whereas Bogart abstracted the code in various ways. As a result, the original assembly-language programmers found it harder to debug Bogart's code. This reaffirms the need for an explanation facility in intelligent applications.

Keywords: Portability; Software migration; Bogart; Abstraction; Transformation; Re-implementation

1. Introduction

As the field of Artificial Intelligence matures, expectations of large-scale industrial-grade applications naturally grow. While AI has an increasing effect in industry, especially in the form of knowledge-based systems, a lot of the research remains in the laboratory, uses toy examples, and does not scale up to large real-life applications. All too often, when theoretical ideas are demonstrated at all, it is in the form of complex programs that require heavy computational resources even for small examples.

The same phenomenon also plagues research in Software Engineering; in this context, Potts [18] describes the more usual methodology, which he calls "research-then-transfer", as research that is carried out in an academic setting with a vague expectation that it will later be transferred to industry, an expectation that is rarely fulfilled. In contrast, Potts argues for the "industry-as-laboratory" methodology, in which research is carried out in industry, using real, large-scale problems. This helps to ground the research in reality, so that it is clear that the problems investigated are not artifacts, and also demonstrates that the solutions are effective and scalable.

This paper describes research carried out using the industry-as-laboratory methodology. It applies Artificial Intelligence techniques to a Software Engineering problem of high practical significance—software migration. This is the problem of porting existing software applications ("legacy software") that are specific to a single platform (hardware, operating system, DBMS, etc.) to other platforms. The existing code often embodies numerous hidden assumptions about its environment; these assumptions need to be discovered and modified in order to allow porting to other environments in which they do not hold. This is most severe in the case of software that is written in the assembly language of a particular machine, and is tailored to the characteristics of that machine.

Specifically, this research tackled the problem of automatically translating a large system written in IBM 370 assembly language to C in order to port it to different platforms such as Unix workstations, AS/400 machines, and even PCs. In the process of developing the translation system, we had to pick and choose from the techniques described in the literature, adapt them to the specific problem, and invent some new ones. The constant need to apply the results to real data focused the research, and taught us important and sometimes unexpected lessons, which are presented at the end of the paper. The company started its own brute-force translation project before we suggested the AI-based solution. A comparison of the two approaches shows that the AI approach requires considerably more resources for development, and requires further support tools, such as an explanation

facility. However, given the right commitment, the AI approach significantly outperforms the typical industry brute-force approach on all criteria.

1.1. The problem

The example used in this case study was a large database system and application-generator written in IBM 370 assembly language, the main product of Sapiens International, Ltd. 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. At the same time, the main product running on IBM mainframes had to be supported and maintained. While high-level-language 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 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 high-level-language code be generated completely automatically from the assembly-language source, in order to allow continuing modification of the latter. Some manual preparation of the assembly-language sources was clearly necessary, since they contained self-modifying code and other non-portable techniques. Further manual preparation for translation was therefore also acceptable, but it was important to minimize the amount of manual effort required. On the other hand, readability of the resulting 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 one or more 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 so-called 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. Many patterns of coding were outlawed, and others had to be replaced by special macros. The result was that the complete sources had to be carefully inspected and extensively modified, requiring a heavy investment of painstaking (and boring) effort by the original programmers.

1.2. The AI solution

The situation at Sapiens was thus an excellent opportunity for testing an Artificial Intelligence approach to the translation problem as well as for comparing it with the typical

industrial brute-force approach. In contrast with the naive translator initially developed by Sapiens, the approach suggested in this work is based on AI principles; it emerges from a model of the way human programmers translate programs, and its goal is an intelligent and extensible translation tool.

It was clear 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 24-bit or 31-bit long), and access to parts of multi-byte entities (depending on byte order within words).

Like many other AI techniques and principles, translation by abstraction was previously only demonstrated on toy examples [7,24]. This work is a first attempt to apply these principles to large-scale programs. In 1993 the development of a system based on abstraction and re-implementation was initiated, and called Bogart. [2]

Bogart is based on the premise that textual format is not a convenient medium for program manipulation. For this purpose, a program is best represented in terms of control-flow and data-flow abstractions. This also seems to fit the way programmers think about their code.

By translating the program into an abstract representation, it is possible to transcend the details of the source language that are irrelevant to the algorithm, perform transformations on the abstract representation, and re-implement the algorithm in the target language. For example, one of the fundamental aspects of assembly language programming, the use of registers, can be abstracted away because registers are only used for temporary storage of values necessitated by hardware limitations.

The results demonstrate that the abstract representation adopted is a powerful tool for automatic translation. It exceeds the translation-by-simulation approach on all criteria: the portion of the source language supported is larger, less manual work is required, the resulting code is more portable, efficient, and readable. Most important, this has been proved in a large-scale real-world project.

Bogart translated a central Sapiens module, which replaced the original module in the Sapiens system to form a working subsystem. This module is twice as fast as the version produced by the simulating translator. The code produced by Bogart for small examples is as fast as code written by a human programmer directly in C. In several cases, Bogart's code is even faster. In terms of code size Bogart has an overhead of approximately one third over manually written code. These results far surpass those achieved by the simulating translator: the code it produces is typically between two times and ten times slower than that produced by Bogart, and is up to twice as large.

Bogart produces more portable code. Small programs translated by Bogart were successfully run on an AS/400 machine, a machine with some esoteric specifications. The code produced by the simulating translator could not run on the AS/400.

Bogart requires less manual work in preparation of the code, since it performs more sophisticated analysis and can handle correctly a greater variety of coding practices. The

[2] Better Optimizing General-purpose Abstract Representation Translator, and also the name of the second author's dog.

intensive manual preparation of the code for the simulating translator was not only time-consuming but also damaged the quality of the original code, produced undesirable results, and seriously damaged staff morale.

The analysis technology that lies at the heart of Bogart has recently been converted to support analysis and remediation of assembly-language programs for the Year 2000 (the famous "millenium bug"). This tool, called Falcon2000 by Sapiens, now forms the base technology for a 100-person factory run by Sapiens, and is responsible for a large part of the company's revenues.

2. Translation by abstraction

2.1. AI techniques

Bogart relies on several AI tools and techniques. These include:

- *The Plan Calculus*: originally developed as the internal representation for the Programmer's Apprentice [20], it forms the basis for Bogart's abstract representation of the code.
- *Translation by abstraction and re-implementation*: this idea was presented in a theoretical framework by Waters [24]. With the addition of program transformations, it serves as the general framework for Bogart, where it is made concrete.
- *Program transformations*: typically performed on text-based representations of code, they are more natural and effective when performed on a more abstract representation. A transformation step was added between the abstraction and re-implementation steps, in order to bring the program closer to the conceptual model of the target language, and to perform some optimizations that are beyond the power of the target compiler.
- *Constraint propagation* techniques were used to compute subroutine interfaces, and to deduce the best types for data values.
- *Pattern recognition* techniques were used to a limited extent in Bogart. They were useful for certain low-level patterns, such as the recognition of simple control structures. However, it was not necessary to recognize more abstract structures, because the target language itself is rather low-level. This was fortunate, since the current state-of-the-art in cliché recognition [26,28] is not sufficient for large-scale legacy applications.

The rest of this section describes the operation of Bogart as it relates to the above principles. More details can be found in Friedman's M.Sc. thesis [8].

2.2. The Plan Calculus

The theoretical framework for this work lies in work done on the Programmer's Apprentice project at MIT's Artificial Intelligence Laboratory by Rich, Waters, and others [20]. The Programmer's Apprentice was intended to be a general-purpose knowledge-based automatic tool with some program understanding abilities, used to assist all programming tasks.

The key insight at the basis of the Programmer's Apprentice project was the observation that when expert programmers think of code they tend to ignore the syntax of the programming language they use. Instead, they focus on higher-level abstractions that are more naturally described in terms of data- and control-flow. This observation was formalized in the Plan Calculus representation for programs [19,20]. This is a wide-spectrum language-independent formalism that can directly express the conceptual building blocks from which programs are built, at various levels of abstraction, while eliminating syntactic means for achieving them (such as local variables and control structures). In the Programmer's Apprentice project, plans were used to support synthesis, analysis, and debugging at various levels of abstraction. For example, Wills' Recognizer [25] produced high-level documentation of a program by first translating it into "surface plans"—detailed representations of the original programs as plans. By using a library of programming idioms, or clichés, as a plan-based grammar, the Recognizer discovered conceptual structures at increasing levels of abstraction, thus re-creating the original design decisions of the writer of the program.

When translating assembly-language programs, significant abstraction is achieved merely by the translation to surface plans. Some of the transformations performed by Bogart abstract the representation further, but no general parsing capabilities were necessary in this case.

We found the Plan Calculus to be a convenient medium for program manipulation; it allowed us to ignore syntactic details from the first stage, and was natural for the transformation and re-implementation tasks. While this observation does not directly support the claim that programmers naturally think in terms of clichés, it certainly affirms the utility of the plan representation from an engineering perspective.

2.3. *Translation by abstraction and re-implementation*

Waters [24] suggested that high-quality translation can be achieved by abstraction to a formalism such as the Plan Calculus followed by re-implementation in the target language. Bogart is based on this framework, with the addition of program transformation steps performed on the abstract representation (see Fig. 1). The transformations are used to achieve further abstraction as well as efficiency of the target code.

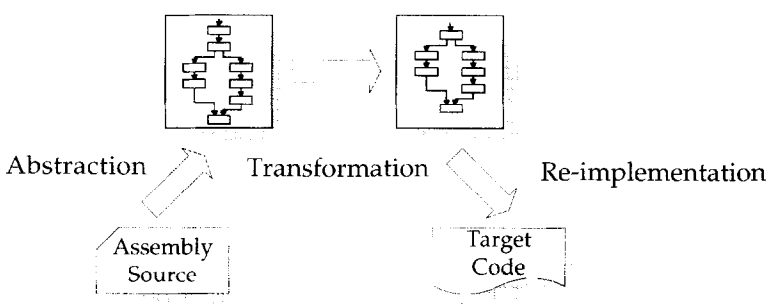

Fig. 1. The structure of Bogart.

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 can 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.

2.4. An example

Bogart's performance can be illustrated by one of the small examples for automatic translation, the Horner routine, taken from an IBM assembly-language textbook [23]:

```
 1 HORNER CSECT
 2        STM  R14,R12,12(R13)
 3        LR   R12,R15
 4        USING HORNER,R12
 5 *      ** INITIALIZATION **
 6        LA    R7,COEF
 7        L     R5,0(R7)        SUM = A0
 8        LA    R9,0            I = 0
 9 *      ** TEST FOR EXIT **
10 LOOP   CR    R9,R2
11        BNL   OUT
12 *      ** ADJUSTMENT STEP **
13        LA    R9,1(R9)
14        LA    R7,4(R7)        NEXT COEFFICIENT
15 *      ** BODY OF LOOP **
16        MR    R4,R3           SUM*X
17        A     R5,0(R7)        SUM = SUM*X + AI
18        B     LOOP
19 OUT    LR    R0,R5
20        LM    R1,R12,24(R13)
21        BR    R14
22 COEF   DS    10F
23        END
```

This routine computes a polynomial using Horner's method. COEF points to an array with the coefficients ordered from a_0 to a_n. The address of the array is loaded into general register 7 (R7) in line 6. Register R5 accumulates the sum, and R9 is the loop counter. Registers R2 and R3 are the routine's arguments: R2 holds the degree of the polynomial, and R3 holds the value of x. The main loop is in lines 10–18, and the routine returns a value in R0. General registers 1–12, 14 and 15 are saved on entry to the routine by the

Store Multiple (STM) instruction, and the values of registers 1–12 are restored before the exit by the Load Multiple (LM) instruction.

The main difficulties of the translation from assembly language can be demonstrated by this small example. Most of the computation takes place in the 16 general-purpose registers; values have to be loaded into the registers and stored after the computation. High-level languages such as C do not use registers, and specify computations in terms of expressions involving storage locations.

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, a portable translation to C must associate a correct data type with each variable.

The brute-force simulating translator generated the following code:

```
 1 void HORNER (tagSAPReg *Reg)
 2 {
 3    T_stm(14,12,((Reg[13].ucp+12)),Reg);
 4    Reg[12].sw = Reg[15].sw;
 5    Reg[7].pv = &(COEF[0]);
 6    Reg[5].sw = *(sWord *)Reg[7].ucp;
 7    Reg[9].sw = 0;
 8 LOOP:
 9    if ((Reg[9].sw) >= Reg[2].sw) goto OUT;
10    Reg[9].sw += 1;
11    Reg[7].sw += 4;
12    T_mult(&Reg[4],Reg[3].sw);
13    Reg[5].sw += *((sWord *)(Reg[7].ucp));
14    goto LOOP;
15 OUT:
16    Reg[0].sw = Reg[5].sw;
17    T_lm(1,12,((Reg[13].ucp+24)),Reg);
18    return;
19 }
```

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, and therefore requires the function `T_mult` for simulation.

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

```
1 sWord HORNER(sWord r2sw, sWord r3sw)
2 {
3     sWord    r5sw;
4     sWordPtr r7swp;
```

```
 5      sWord    r9sw;
 6      r7swp = (sWord *)(&COEF[0]);
 7      r5sw = *r7swp;
 8      r9sw = 0;
 9      while (r9sw < r2sw) {
10         r9sw++;
11         r7swp++;
12         r5sw = r5sw * r3sw + *r7swp;
13      }
14      return r5sw;
15  }
```

This example demonstrates several important ways in which Bogart produces shorter, more efficient, and more portable code:

- *Removal of redundant code.* Some assembly instructions do not have to appear in the high-level code at all. For example, lines 2–3 and 19–20 of the original code have to do with OS/370 calling conventions and register save-areas, which are irrelevant to other architectures and are supplied automatically 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 (line 12):

  ```
  r5sw = r5sw*r3sw + *r7swp;
  ```

  instead of the two statements:

  ```
  T_mult(&Reg[4],Reg[3].sw);
  Reg[5].sw += *((sWord *)(Reg[7].ucp));
  ```

  generated by the simulating translator (lines 12–13).
  Not only does combination of expressions greatly enhance readability, it also has a great impact on the performance of the translated code. When registers are simulated by means of an array, each operation involves access to memory. C compilers typically cannot optimize a sequence of operations into the equivalent of a single expression, since apparently different memory references may potentially collide (this is the "aliasing problem").[3] Since we know that registers cannot be aliased to each other or to memory, we can perform this optimization in the translation, thus allowing the C compiler to perform further optimizations in the translation back to machine code.
- *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

[3] Optimizing C compilers usually have an "escape clause" that causes them to ignore aliasing. This can solve the optimization problem, at the expense of potentially introducing subtle compilation errors into the resulting code.

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 program.

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 simulating function 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 (as in line 12).

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. The IBM 370 architecture (like many other CISC machines) contains a number of further instructions that generate multiple results. (Indeed, this technique is particularly valuable in the timely problem of CISC to RISC migration.)

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 program, 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. The "code improvement" document required the manual detection and eradication of such violations; the simulating translator is not even able to detect them.

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;
```

Such cases are not uncommon in legacy code; assembly-language programmers had been taught to mix unrelated instructions in order to achieve the best use of the machine's pipeline.

- *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 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 currently performs mainly local type analysis. 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++` (line 11) instead of the simulating translator's `Reg[7].sw += 4` (line 11). More details are presented in Section 2.7.
- *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 easily recognized by Bogart.

2.5. *Program transformations*

A transformation step was added between the abstraction and re-implementation steps. The transformations are applied to the internal representation generated by the abstraction step, and serve to generalize and optimize it, and prepare for more readable code.

For example, one transformation is responsible for removing over-constraining control-flow edges. In a single control block, it is possible to reorder computations that do not have side effects. Bogart removes unnecessary control-flow edges, making it possible to collect expressions from non-adjacent source lines.

Another transformation removes code segments that are not needed for the translation, such as code implementing OS/370 calling conventions and code that sets up base registers for code and memory addressability.

In general, Bogart avoids transformations that duplicate code. One case in which code is duplicated is when the condition code is tested twice, and the predicate being tested does not involve a side effect or a lengthy computation. For example, in the code fragment from

the module GREDCE shown earlier, the two tests of the condition code (`BH` and `BNH`) were converted into two separate tests (`r7sh > temp` and `r7sh <= temp`) instead of setting a "condition code" variable and testing its value.

2.6. *Discovery of subroutine interfaces*

Various conventions can be used for parameter passing in assembly language. The IBM 370 is not a stack-based machine, so arguments have to be passed (directly or indirectly) through the machine registers. IBM's operating system, OS/370, dictates a convention for parameter passing and saving the values of registers, but this is typically only followed at module boundaries. Internal subroutines usually employ idiosyncratic conventions about passing arguments and results in registers, and about which registers are saved across calls.

Since a major part of Bogart's effort is aimed at abstracting away from the use of machine registers, it was necessary to recover these conventions for each internal subroutine, and declare interfaces accordingly. This was done using constraint-propagation techniques on the abstract representation. As preliminary steps, all values that are not used at all are eliminated, and values that are preserved across a subroutine call (either because they are not changed inside the subroutine, or because they are saved on entry and restored before exit) are short-circuited in the calling routines.

After data-flow analysis is performed, values are traced backwards to their origins across routine boundaries. This makes it possible to identify potential inputs as values that are used inside a subroutine but not generated by it, and potential outputs as any values generated inside the subroutine and present in the registers on exit. The interface discovery process identifies exactly those potential inputs and outputs that are really necessary. It does this by considering the projection of the data-flow graph on all potential inputs and outputs, and marking values as "live" if they are actually used in the computation.

Any potential input that has a data-flow successor that is used inside the subroutine to effect control-flow splits or changes to memory is initially marked as live. Any data-flow predecessor of a live value in the reduced graph is itself marked as live. The propagation process continues until no further change occurs. At this point, all potential inputs and outputs that are marked are considered to be part of the routine's interface. In this way, values that are generated for internal use only are removed from the interface. This technique is general enough to cover the case of mutually recursive subroutines.

2.7. *Discovery of data types*

Assembly-language programs contain little information about data types, and the information is not always correct. Sizes are usually (but not always) declared correctly, but the rudimentary types supported by the assembler are many times misused. For example, the idiosyncrasies of the IBM 370 assembler force programmers who wish to initialize storage using constant expressions to declare them as addresses (provided they are small enough to be used as pointers). Portable translation to a strongly-typed language such as C requires more specific data types.

Bogart currently computes types only for values in registers, although the same techniques can be applied globally to compute types for storage variables as well. The

computation of data types is based on the propagation of type information from actual usage of values. For example, an Add Halfword operation indicates that its operand is a halfword (16 bits), even though operations on general registers usually involve all 32 bits. The type of operation can also give partial information about the type of the value: for example, a value that is de-referenced must be a pointer, and a value that is used in an integer addition could be an integer or a pointer. Types can also be propagated across operations. For example, the addition of two integers yields an integer, while the addition of an integer and a pointer yields a pointer. Once the type of a de-referenced value is known, the type of the pointer can also be deduced.

For the Year 2000 problem, this analysis was further enhanced in order to reveal precise date formats (YY, YYMMDD, etc.). This did not require a change in the algorithm, only the addition of more specialized constraints.

Type information collected from such sources is propagated through the data-flow network. Each value is associated with a set of possible types, which shrinks as the result of information propagated from neighboring nodes. In this way, the type information is refined until the best possible type is computed.

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. The module consists of about 4500 lines of code.[4] 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 assembly-language textbook [23]. 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 simulating translator on all counts: assembly language coverage, the manual work involved in preparing the code, the target code's portability, and code efficiency.

Another criterion in the comparison is the possibility for future enhancements of the translators. The only method that will allow the simulating translator approach to produce more portable code is by additional manual preparation. In contrast, the abstraction approach can be improved by more sophisticated analysis, such as cliché recognition. The advantages of the more general technology were strikingly demonstrated by the adaptation of Bogart to Falcon2000, the Year 2000 analysis and remediation tool, done more than a year after the development of Bogart was complete. The initial adaptation required only five weeks.

As mentioned in Section 1.1, maintainability of the target C code was not important for this translation effort, since maintenance was intended to be done on the original assembly sources rather than on the translated C code. For this reason, relatively little effort was

[4] This is about average for Sapiens modules, and is larger than most modules encountered by the Year 2000 factory.

spent on readability in Bogart. Still, it is evident that Bogart's code is much more readable than the code produced by the simulating translator. However, more work is necessary in order to produce more readable code. (See Section 3.6 for a discussion of the issue of debugging the target code.)

3.1. Portability

As expected, Bogart produced more portable code than the simulating translator. The code produced by the simulating translator 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 can distinguish finer types, e.g., between integers and pointers, and can therefore produce code for a much larger variety of architectures. We will examine the portability of the generated code by discussing some of the other target platforms, which usually have different hardware specifications. In this section we present two major issues representing such differences.

First, it is possible to have one instruction generate different results, depending on the type of the operands involved. Without type analysis, it is impossible to determine correctly the intended result of an instruction. As an example we can pick any instruction that looks at only a part of a word, or changes only a part of a word. Consider the following simple assembly code:

```
L  R5,X
IC R5,A
ST R5,Y
```

The Insert Character (IC) instruction inserts one character into the lower byte of a register. The C code produced by the simulating translator is:

```
Reg[5].sw = X;
Reg[5].uc[BThree] = A;
Y = Reg[5].sw;
```

The symbolic constant `BThree` is defined according to the byte order on the target machine. This, however, is not enough, since the type of X is unknown. If X is an integer the translation would be correct, because `BThree` would correspond to the lower byte. If, however, X is a character array, the wrong byte would have been changed. In this case, the correct index should be the constant 3, regardless of the byte-order of the target architecture.

This example demonstrates that there might be simple assembly code that cannot be translated by the simulation approach.[5] In contrast, by knowing the type of the variable in register 5, it is easy for Bogart to produce the right code for each type, for each target machine.

[5] It is theoretically possible to solve such problems by simulating each instruction at the machine level (using big-endian byte order for all operations and variables, regardless of the byte order of the target machine). However, simulation at this level incurs a considerable loss of efficiency and was not considered in this case.

A second major portability problem involves porting the code to systems with non-flat memory[6] or non-word pointers. The simulation heavily relies on pointers behaving like integer words. Any operation on pointers treated by simulation as integers, and vice versa, may cause errors. This is far too frequent to allow manual handling. This problem prevented a test program translated by the simulating translator from running on the AS/400, which has 128-bit pointers. The same program translated by Bogart ran successfully, because there was no ambiguity between pointers and other types. However, we stress that this was a small program. Further sophistication is necessary for Bogart to be able to produce AS/400-compatible code from large programs; in particular, more accurate type analysis will probably be necessary.

3.2. *Efficiency*

Bogart produced much more efficient code than the simulating translator, in terms of both space and time. Typically, Bogart code was between 25% and 50% smaller and more than twice as fast as the simulating 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 remove some inefficiencies 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 assembly-language textbook [23]). 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 results are normalized so that Bogart's code is the unit of comparison in each case.)

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.

Also 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

[6] Non-flat memory here stands for segmented memory systems, such as that of the Intel 80x86 family.

Table 1
Results of translated programs on RS/6000

Module	Simulating translator output		Bogart output	
	time (s)	space (bytes)	time (s)	space (bytes)
SAPDBMS	18	41700	9	29073
Bin	63	4170	33	2802
Horner	10	3302	3	2465
Random	9	5447	4	2741

Table 2
Relative running time of BIN program on various platforms

	IBM 370	RS/6000	AS/400	Microsoft C	Borland C
Original (assembly)	0.33	—	—	—	—
Simulating translator	1.74	1.91	*failed*	1.31	1.20
Bogart	1	1	1	1	1
Hand-crafted C	0.91	0.97	0.49	1.03	1.11

not test the translated SAPDBMS module on the IBM mainframe, but we expect similar results. (Some recent comparisons appear in Section 3.7.)

3.3. Assembly language coverage

Bogart performs deeper analysis of the assembly-language source code, and therefore places less stringent demands on the source. One example (which was discussed earlier) is the data-flow analysis of the value of the condition code, allowing arbitrary placement of branch instructions. (The simulating translator does not allow testing the condition code after a successful branch.) Another example is the use of the high bit of pointers. In the IBM 370, this bit is not part of the value of the pointer and is often used as a flag. This practice had to be manually removed from the Sapiens code—a task requiring careful examination of the code. Since Bogart identifies pointers, it could discover such uses and change them automatically. [7]

Many cases in which the simulating translator required special macros to be placed in the source could have been handled by Bogart without any change. For example, a subroutine returns to its caller by a Branch Register instruction, which has other uses, such as jumping through a branch table or a "computed goto". The simulating translator cannot handle this instruction, and requires that subroutines be modified to use a special macro for returning. Since Bogart performs data-flow analysis on registers, it can identify the source of the value in the register used for branching, and discover those cases in which the Branch Register instruction is used for returning.

[7] This was not done, because Bogart used sources already prepared for the simulating translator.

Certain assembly-language programming techniques, such as self-modifying code, have no parallel in high-level languages and are even bad practice in assembly language. These could not be supported by either translator. Still, there is the question of identifying them automatically. In this respect, Bogart well exceeds the simulation approach.

3.4. Typing issues

The simulating translator relies on user definitions of variables, which include the types of variables. Currently, Bogart uses this information, but it has an advantage over the simulating translator: it is able to trace values and to know the types in the registers. Also, Bogart traces use of values and can deduce simple but important information, such as whether a value is a pointer or not.

The simulation approach dictated standards requiring manual modification of the code: for example, the assembly types A, F and X are supposed to correspond to a void pointer, an integer, and a character, respectively. This requires manual work by the assembly-language programmers in reviewing and changing the data definitions. On the one hand, it is reasonable to invest in manual work at the data definitions rather than at the code areas, because they are much shorter. On the other hand, the information is partial and cannot be automatically verified. Practice showed that manual work that could not be automatically verified produced some of the most elusive errors.

The simulating translator lacks knowledge of the type of the value in a register. Thus, the standardization demands grew stricter than originally planned, and the translator issued a lot of spurious warning messages. In practice, some modules were rewritten using only a small part of the available assembly-language instructions and constructs, and in other cases the warning messages were ignored.

Bogart can correctly translate code that does not comply with these strict standards. However, such code was not thoroughly tested so far. The important test for portability is trying to compile and run target sources on platforms with system specifications different from those of the IBM 370. The code produced by the simulating translator, with most warnings ignored as they are today, is not expected to run on such systems. Some of these problems can be solved by Bogart today, as has been demonstrated on small programs. On Sapiens code, Bogart currently uses sources already prepared for the simulating translator, and thus its support for these constructs is similar to that of the simulating translator.

3.5. Manual preparation

Translation by abstraction requires less manual work than translation by simulation, 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 simulating translator, it is impossible to quantify the difference, but it is clear that several of the "improvements" necessary for the simulating 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. (Global data-flow analysis may help to solve this contrived problem.)

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 versions that coexist with the original assembly versions. 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.

3.6. Debugging the target code

There are two distinct modes of working with automatic translation. It can be used as a one-time effort to translate a large piece of code, which will then be debugged and maintained independently. This requires the target code to be readable and properly documented. The Sapiens case is different. One of the objectives achieved by automatic translation is having two working versions of the same product. It was desired that for an intermediate period of time, which may last for several years, the development and maintenance will be done on the assembly-language source, and nothing will be done on the target code (except for verifying that it works).

The two modes imply different sub-goals for the translation. In the second case, readability is a minor issue. Debugging of the result code is done by the assembly-language programmers. Thus, it is more important for the original assembly code to be reflected in the C code, than for the code to have C-like appearance.

The simulating translator preserved the structure of the original program, whereas Bogart abstracted the code in various ways. As a result, the original assembly-language programmers found it harder to debug Bogart's code. Certain allowances were made in Bogart for ease of debugging; for example, temporary variables created by Bogart are named after the registers containing these values in the original code (see the example in Section 2.4). However, more is needed; a full solution will have to include a comprehensive explanation facility that will help the programmers understand the relationships between the C code and the original source. This is consistent with conclusions reached in the development of other intelligent applications, in which increasing sophistication is accompanied by greater demands for intelligent explanation.

A step in this direction has already been taken in the conversion of Bogart for the Year 2000 problem. This tool, Falcon2000, displays the source with various annotations about variables and values suspected to contain date-related values. Using a simple graphical interface, it is possible to query each suspect in order to find out what is the reason to suspect it. This, in effect, presents a mission-oriented trace of the data-flow and control-flow graphs of the program. Through the use of Falcon2000, we have also

discovered the need to provide dependency-directed backtracking mechanisms for making incremental changes in the status of suspects. If Bogart is enhanced with analysis tools that require manual supervision, it will need similar mechanisms.

3.7. Other results

Falcon2000, the Year 2000 analysis and remediation tool based on Bogart, has by now been used to analyze over 30 million lines of assembly code. The tool tracks potential date problems, and suggests how to fix the code. The modifications were done under human supervision, but results show that for a large portion of the modules analyzed, the tool would correctly convert the whole module. It should be noted, though, that this task is easier than translation to C, since it only requires a narrow view of the program.

Sapiens is now considering the development of an assembler-to-C translation product based on Bogart, to be called Falcon-C. This tool will have to address, among other things, the readability of the target code. Preliminary tests were performed on assembly-language programs written for TPF, an IBM/370 operating system for high-volume transaction-processing systems. A complete small package (consisting of six modules of less than 500 lines each) was translated. The running time of the converted code was very close to that or hand-crafted code, and was only 25% slower than the original code.

4. Conclusion

4.1. Related work

4.1.1. The Programmer's Apprentice

Several translation systems were designed as part of the Programmer's Apprentice project. In the case of Faust's Satch system [7], the source language is Cobol and the target language is Hibol (Hibol is a very-high-level non-procedural business data processing language). The motivation was to convert pre-existing Cobol programs into a form where they can be more easily maintained.

Satch is similar to Bogart in that the target language is on a higher level than the source language. These cases are, according to Waters [24], the hardest for translation, and impractical for translation by simulation. As a result of this similarity, Satch is also similar to Bogart in that much work is done in abstraction of the source, while code generation is relatively straightforward.

Satch includes sophisticated components such as temporal abstraction and algorithm recognition. However, Satch is only a demonstration system, and has only been tested on a few example programs.

Another system, Duffey's proposed Cobbler system, was discussed by Waters [24]. It uses translation via abstraction and re-implementation in order to compile Pascal programs into PDP-11 assembler language. Cobbler's goal is the creation of extremely efficient object code—comparable in efficiency to that produced by an expert assembly language programmer. Cobbler is expected to achieve this goal by changing the program's algorithm. Cobbler was designed, but no effort was made towards its implementation [24].

Satch lacks knowledge-based cliché recognition. This was the objective of Wills' Recognizer [25,26]. The Recognizer, which is the most ambitious use of the Plan Calculus for abstraction, was targeted as a program understanding tool. It employs a technique in which program recognition is treated as a parsing task. The basic idea was to convert the program into a graph representation (which evolved from the Plan Calculus), translate the library of familiar structures to be recognized into a graph grammar, and then parse the program in accordance with the grammar, using Brotsky's graph parsing algorithm [4].

The Recognizer was tested only on toy programs, and was able to demonstrate in-depth understanding of these programs. However, real-world programs are rarely made up entirely of familiar forms and data structures. Parsing of large programs has not been shown practical and cost-effective in real life. In our case, the concept of abstraction is suitable, since assembly language is on a lower level than C. Nevertheless, the level of abstraction required from Bogart is limited: only those abstractions directly expressible in the target language need to be found. Wills' clichés would not consist of more than a few percent of our sources, and in our case cliché recognition was not necessary.

Wills has extended the Recognizer to a program called GRASPR [27,28], which has been used to analyze larger programs (500–1000 lines) from real applications, although these had to be manually transformed into a pure functional form first. GRASPR can also recognize abstract data types and aggregations. Data structures play an essential and synergistic role in program understanding. If such knowledge exists, then a more significant part of real-life programs, such as Sapiens source, may be parsed as clichés, and applied to user-defined data types.

4.1.2. Environments for assembly languages

The research described above is ambitious in its goals and is far beyond what can currently be achieved with real large-scale code. A more modest goal is to provide an environment that will assist a human reengineer. An example is the Maintainer's Assistant [2], a transformation system for reverse engineering. The Maintainer's Assistant first translates its input in a literal fashion to an internal wide spectrum language (WSL), which is still text-based. Then it allows the user to choose transformation to apply to the code. The result could in principle be re-implemented in a conventional language, although this is not described in [2]. This system has been used on real IBM 370 assembly code, with some reduction in size (20% in the example cited) in the translation to WSL. This is disappointing, since WSL is a more abstract language, and more could be expected. This is probably due to the rather low-level transformations provided.

A more recent effort is Mandrake [15], a tool for reverse-engineering IBM 370 assembly-language code. Mandrake is similar to Bogart in that it translates its source code to a wide-spectrum representation, and performs transformations on that representation.[8] In contrast with Bogart, which uses the Plan Calculus for the intermediate representation, Mandrake uses abstract syntax trees of a C-like language. This is a less abstract representation, and seems to make high-level transformations more difficult.

The goal of Mandrake is to assist the reverse-engineering of legacy code by "producing a 'draft' of a high-level language version, to be verified, modified, and polished by competent

[8] The final stage of re-implementation in C or another real language has not been implemented.

software reengineers". Readability of the code produced by Mandrake is therefore more important than absolute correctness. This suggests that more tools (for explanation and analysis) will be found to be required if this system is put to actual use.

Because its output is only a draft, Mandrake places more importance on readability of the final code than does Bogart. Its major technique is to replace unstructured jumps with exits from the middle of blocks. This seems insufficient for the generation of quality code, and indeed the code given in the example in [15] does not appear to be more readable than Bogart's code. Recognition of higher-level clichés that could enhance readability, such as access to a matrix with two indices, is not performed.

Mandrake has not been tested on real large-scale code. In fact, because of the assumptions it makes on the style in which its input code is written, the authors say that "the practical goal of the project was to accurately translate programs of the stylistic quality found in case studies contained in a [textbook] on IBM assembler programming".

A more practical (but more limited) tool is PARE [21], a reverse engineering environment for assembly languages, distinguished by being based on a language-independent substratum. PARE analyzes the input program and presents various views of it to the reengineer. The level of analysis performed by PARE is relatively low, and it could be enhanced considerably using the techniques described in this paper. Conversely, a tool like PARE could be useful in the manual pre-processing stage required for automatic translation by Bogart.

Another assembly-language analysis tool is REAP [13], which translates its assembly source code to an intermediate formalism called XANDF. REAP has a flexible hardware model, and can explicitly represent the assumptions for neglecting certain features of the real hardware. Unlike Bogart, REAP is a tool that needs to be manually controlled.

Recently several companies have started advertising translation systems of various kinds, including assembler-to-C. From the available information it seems that these tools are similar to the brute-force literal translator, with little or no further analysis.

Cifuentes [5] describes the design of an integrated environment for reverse engineering programs originally written in a high-level language from their binary format. Like Bogart, this environment uses a language-independent intermediate representation. It includes an "idiom analyzer", but the idioms it detects are those generated by the compiler (for example, using two 16-bit subtractions to implement 32-bit subtraction on a 16-bit processor).

4.1.3. Reverse engineering

Software migration may be considered an example of the more general task of software reengineering. Early work in the 1960s discussed restructuring programs containing the notorious GOTO statement. Only in the late 1980s did the term "software restructuring" give way to "reengineering" [1]. The feasibility and cost-effectiveness of legacy-software reengineering remains a theme for debate.

Harandi and Ning [9] suggest a maintenance-support system, which supports activities such as documentation, correction, and enhancement. PAT (Program Analysis Tool) "uses an object-oriented framework to represent programming concepts and a heuristic-based concept-recognition mechanism to derive high-level functional concepts from the source code" [9, p. 74]. PAT explicitly represents not only programming knowledge but also

analysis knowledge. PAT also uses a truth-maintenance mechanism, to allow for the information about the program to be updated when the program changes. However, PAT was not applied to real-world programs.

Hausler et al. [10] also suggest a system for program understanding based on abstraction. Their algorithms apply only to structured code. Also, no real-world attempt was made.

Current tools focus on identifying components and producing high-level descriptions. For example, Ning et al. [16] suggest the concept of *reusable component recovery*. Both goals, recognizing high-level patterns and producing high-level documentation, were not necessary in our case. Component recovery sometimes requires a deep level of code understanding that is beyond that necessary for assembly-to-C translation, and is more difficult to achieve. In consequence, such tools are not automatic, and need manual supervision.

Ning et al. discuss Cobol/SRE (Cobol System Renovation Environment), a set of tools for identifying and extracting components from large Cobol legacy systems. The tools include program text browsing, flow analysis, complexity analysis, and program segmentation. In addition, it includes a tool for concept recognition: a construct is encoded by the programmer as a plan and searched in the code. The authors admit that this component's contribution is limited. They focus on the component recovery tool. This tool is not automatic, it is a convenient environment to aid a human programmer.

4.1.4. Empirical research

One approach to an Artificial Intelligence solution to program translation would first address the question of human program comprehension. This work did not aim at an empirical study of human programmers, but several such efforts appear in the literature. The assumptions in this work about programmers generally agree with the views taken in those papers.

Soloway and Ehrlich [22] tried to empirically prove that experts' understanding of programs depends on two main factors: the extent to which the program is comprised of familiar plans, and the extent to which it complies with a set of discourse rules. Programs are composed of plans; the composition of these plans into programs is governed by rules of programming discourse. Thus a program may be correct in solving a problem, but yet disobey the discourse rules (e.g., by having variables with misleading names), and thus be difficult to understand by human experts.

The idea of programming plans is central to the approach of this work, even though high-level clichés were not automatically recognized. No attempt was done in any of the works surveyed to use the programming discourse rules. These rules may suggest heuristic methods for recognizing plans in programs, if proven feasible to automate.

Empirical research also strengthens the point of view taken in this work by stressing abstraction levels (e.g., Brooks [3]). However, there is no reason to suggest that the levels of abstraction assumed in this work are identical to the abstraction levels held by human programmers. Also, the abstraction process is described as being employed by humans in a top-down strategy, unlike the bottom-up approach used by Bogart and most other automatic systems.

4.2. Discussion

This work is an attempt to apply methods and techniques taken from research in Artificial Intelligence to an acute problem of Software Engineering in a large-scale real-life setting. It has been shown that this approach, when applied carefully, can indeed be superior to the typical industry brute-force approach. However, its limitations were also exposed.

The main advantage of the translation-by-simulation approach is its simplicity. The core of the simulating translator was implemented in three months by one programmer. The corresponding part of Bogart required 30 person-months. Another surprising advantage of the simulating translator was the fact that debugging the code it generated was easier for the original programmers because it was closer to the assembly-language sources. While explanation facilities for Bogart can alleviate this problem, the fact remains that the better it does its job, the harder it is to understand the relationship between its input and output.

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 in the long term, and only if enough resources are allocated. Bogart is now about to produce a Sapiens version that is twice as efficient as that produced by the simulating translator, and more portable. 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 simulating 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. The greatest financial benefit to the organization came not from the translation for which Bogart was originally designed, but from the use of the underlying analysis technology for solving the Year 2000 problem. Based on this success, Sapiens is now considering further development of Bogart to a translation product. 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 in this case, but companies hesitate to invest large amounts of 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 initially had little interest in automatic translation per se, and was only interested in the best translation possible in terms of target quality and investment of resources. The cooperation was convenient for both sides, and we had full access to the data, services from the simulating 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 to be a case of successful cooperation, but unfortunately such opportunities are rare.

A large part of the development of Bogart was dedicated to many theoretically-unimportant details, such as supporting most of the IBM 370 instructions. 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.

Because of the unique nature of this project, we had to be selective in the choice of AI methods to use. Most important was the internal representation for programs. This was based on the Plan Calculus, and demonstrates for the first time its efficacy in a real large-scale setting. However, the full capabilities of the Plan Calculus were not used in this case. The target of the translation is the C language, which is a relatively low-level language. It does not include high-level clichés that require special recognition in the source. Thus, the transition from assembly language to C can be achieved in a very large part by the principle of abstraction, with the addition of the automatic computation of types. High-level construct recognition in the spirit of Wills [25,28] was not useful or necessary. For this reason, Bogart uses a first-principles approach rather than relying on a knowledge-base of programming idioms.

Human programmers use several reasoning techniques when trying to understand an algorithm from the code. They ignore details of the programming language. By trying to understand the roles of the variables, they usually recognize some programming idioms. In this they might be assisted by meaningful labels and documentation. The contribution of these different activities to the translation task is not equal. Also, the relative difficulty of their formalization and implementation in an automatic system are different. Abstraction from language details and identification of variable roles (including their types) were the most helpful in this work.

The ways programmers exploit meaningful labels and documentation is particularly difficult to formalize. The previously-mentioned reasoning forms require knowledge of one domain: programming. This includes knowledge of general programming constructs, data types, algorithms, and programming clichés. However, to use the meaning of names one must have understanding of technical terminology, domain knowledge, and some form of natural language understanding, which is difficult in itself and is beyond the scope of this work. The same argument applies to documentation. Moreover, variable names and documentation are not necessarily correct. Another kind of knowledge that might be useful to a human programmer is an understanding of the program's goal. This knowledge, too, is not available to Bogart.

From the Software Engineering perspective, this research demonstrates that it is possible to achieve software migration commercially by automatic translation. Although some manual preparation of the code is necessary, this differs in spirit from other approaches, such as the Maintainer's Apprentice [2], Mandrake [15], and PARE [21], which are tools for use by a human reengineer, and Ning et al. [16] and Markosian et al. [14], which focus on tools that aid programmers in recovering reusable components. Thus, Bogart shifts the emphasis from user-guided translation to automatic operation. This also implies that correctness of the translation is crucial, since no post-processing is to be done.[9] This differs from tools such as Mandrake, which subordinates correctness to readability.

One issue of assembly-language programming that was largely ignored by Bogart is the use of macros. Sapiens code does not use macros heavily, and Bogart works on the expansion of whatever macros there are. In our investigation of TPF programs (Section 3.7) we found heavy use of elaborate macros. In this case, translating the expanded code would

[9] However, it is possible to make certain assumptions about the original code; if these are not true, they can be established during pre-processing.

create unnecessarily large and obscure code. Conversely, understanding the meaning of the macros could enhance the analysis of the source code.

One approach for dealing with macros is to treat them as extended machine operations. This requires manual analysis of the macro libraries and coding their intent in a suitable formalism. In the case of frequently used macros (as we saw in the TPF code), this effort could be justified. The problem of creating a tool that analyses the macro language and helps in this task is an interesting one, and we may have to build such a tool to be part of a commercial assembly-to-C translation product.

An important aspect of translation by abstraction is its generality. We believe that the same approach could be used to translate other assembly languages. Other architectures have different characteristics that require special treatment. Examples are:

- *Stack-based architectures:* many machines have a hardware stack that is used for subroutine calling (including parameter and result passing) and saving temporary results. Data-flow analysis, as well as the analysis of subroutine interfaces, must take the stack into account.
- *Segmented memory:* some processors, such as the Intel 80x86 family, address memory through pointers composed of two parts: a segment and an offset within that segment. The segments of memory references are typically kept in a special set of registers; offsets are taken from different registers, which may be loaded independently. Sometimes, pointer arithmetic involves unusual carry operations between the offset and the segment.[10] These must be taken into account in data-flow analysis and the identification of pointers.
- *Carry bit:* in order to support multi-precision arithmetic (often on processors whose native word size is 16 bits or less), the hardware has a Carry bit in a special register, and instructions such as Add With Carry and Subtract With Borrow. This simple concept is missing from most high-level languages, and therefore patterns of multi-precision arithmetic have to be discovered and translated appropriately.

In spite of these differences, the general framework of abstraction, transformation, and re-implementation should still be applicable. Most affected, of course, is the abstraction stage, but it should be possible to develop a (relatively small) set of strategies that will cover most common assembly languages.

4.3. Extensions and future work

This research has laid the foundations for automatic translation from a low-level language to a higher-level one. Bogart has achieved significant results, but much still remains to be done. In particular, the next step would be the addition of more global analysis: both data-flow and type analysis over storage locations. This is complicated by the fact that Sapiens code uses large common data areas for inter-module communication as well as for local storage, without clear demarcation. In many cases, the data definitions of these areas are similar but not identical.

Such global analysis could set the stage for the discovery of more complicated idioms (though not necessarily a general recognition component). Examples of constructs that

[10] For example, in real mode on the 80x86 family.

could usefully be identified are the construction and use of parameter lists, and complex control structures such as computed goto's. Identification of complex data structures could be useful for generating more readable and maintainable code, and are necessary for meaningful translation into higher-level languages, and object-oriented languages in particular.

As mentioned above, an explanation facility that will allow programmers to understand the relationship between the original source and the translated code was found to be necessary. Such a component will be even more important as Bogart's abstraction capabilities are further enhanced. By recording the dependency information in Bogart's abstraction, transformation, and re-implementation steps, it should be relatively easy to produce a versatile explanation facility (I-Doc [11] is an example of research in this area). This could form the basis for an intelligent debugging tool, which would assist the programmer by showing data and control dependencies between parts of the code. (Such information would be useful for any debugger, regardless of the translation aspects.)

The general framework described in this paper has been applied to a different domain by Cohen and Feldman in a system called MIDAS [6]. This system converts legacy programs originally written to work with network databases to programs that use relational databases. Specifically, MIDAS converts database access instructions embedded in Cobol programs together with some of the surrounding code into embedded SQL statements. One-to-one translation has been theoretically described in the literature [12], and recently a working system has been announced [17]. However, such translation suffers from the worst of both models. Relational databases compensate for their inherent inefficiency compared to network databases [11] by performing more complex searches and other operations internally. This allows them to optimize these operations by the use of indexes and compilation techniques. Furthermore, in client-server frameworks, much less data is sent over the network. However, programs written for network databases naturally do not take advantage of these capabilities.

MIDAS is also based on the abstraction, transformation, and re-implementation paradigm. It extends the Plan Calculus with a formalism called Query Graphs, which represents database operations. MIDAS uses temporal abstraction to peel as much as possible from the loops surrounding the original database access statements, and then translates them into SQL search operators such as filters and joins and into aggregative operations. In this way we can translate the relevant parts of the original Cobol program into SQL, without having to achieve a complete understanding of the program. The complexity of this analysis depends on the level of nesting of loops in the original program, and we therefore expect it to work efficiently for large programs.

Acknowledgements

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. L. Joskowicz, S. Tyshberowicz, and A. Yehudai provided useful comments on this paper.

[11] This is a result of their greater expressive power.

References

[1] R.S. Arnold, Software reengineering: A quick history, Comm. ACM 37 (1994) 13–14.

[2] B. Bennett, T. Bull, H. Yang, A transformation system for maintenance—turning theory into practice, in: Proc. Conference Software Maintenance, IEEE Computer Society Press, Silver Springs, MD, 1992, pp. 146–155.

[3] R. Brooks, Towards a theory of the comprehension of computer programs, Internat. J. Man-Machine Studies 18 (1983) 543–554.

[4] D.C. Brotsky, An algorithm for parsing flow graphs, Technical Report 704, MIT Artificial Intelligence Lab., Cambridge, MA, 1984 (M.Sc. Thesis).

[5] C. Cifuentes, Partial automation of an integrated reverse engineering environment of binary code, in: Proc. 3rd Working Conference Reverse Engineering, Monterey, CA, 1996, pp. 50–56.

[6] Y. Cohen, Y.A. Feldman, Automatic high-quality reengineering of database programs by temporal abstraction, in: Proc. 12th IEEE International Conference Automated Software Engineering, Incline Village, NV, 1997, pp. 90–97.

[7] G. Faust, Semiautomatic translation of COBOL into HIBOL, Technical Report 256, MIT Lab. for Computer Science, Cambridge, MA, 1981 (M.Sc. Thesis).

[8] D.A. Friedman, Portability by automatic translation: A large-scale case study, M.Sc. Thesis, Department of Computer Science, Tel Aviv University, 1995.

[9] M.T. Harandi, J.Q. Ning, Knowledge-based program analysis, IEEE Software 7 (1) (1990) 74–81.

[10] P.A. Hausler, M.G. Pleszkoch, R.C. Linger, A.R. Hevner, Using function abstraction to understand program behavior, IEEE Software 7 (1) (1990) 55–63.

[11] W.L. Johnson, A. Erdem, Interactive explanation of software systems, in: Proc. 10th Knowledge-Based Software Engineering Conference, Boston, MA, 1995, pp. 155–164.

[12] R.H. Katz, E. Wong, Decompiling CODASYL DML into relational queries, ACM Trans. Database Systems 7 (1) (1982) 1–23.

[13] T. Lake, T. Blanchard, Reverse engineering of assembler programs: A model-based approach and its logical basis, in: Proc. 3rd Working Conference Reverse Engineering, Monterey, CA, 1996, pp. 67–75.

[14] L. Markosian, P. Newcomb, R. Brand, S. Burson, T. Kitzmiller, Using an enabling technology to reengineer legacy systems, Comm. ACM 37 (5) (1994) 58–70.

[15] P. Morris, R.E. Filman, Mandrake: A tool for reverse-engineering IBM assembly code, in: Proc. 3rd Working Conference Reverse Engineering, Monterey, CA, 1996, pp. 57–66.

[16] J.Q. Ning, A. Engberts, W. Kozaczynski, Automated support for legacy code understanding, Comm. ACM 37 (5) (1994) 50–57.

[17] W. Polak, L.D. Nelson, T.W. Bickmore, Reengineering IMS databases to relational systems, in: Proc. 7th Annual Software Technology Conference, Salt Lake City, UT, 1995. Published on CD-ROM.

[18] C. Potts, Software-engineering research revisited, IEEE Software 10 (5) (1993) 19–28.

[19] C. Rich, A formal representation for plans in the Programmer's Apprentice, in: Proc. 7th International Joint Conference Artificial Intelligence (IJCAI-81), Vancouver, British Columbia, 1981, pp. 1044–1052. Reprinted in: M. Brodie, J. Mylopoulos, J. Schmidt (Eds.), On Conceptual Modelling, Springer, New York, NY, 1984, pp. 239–270; and in: C. Rich, R.C. Waters (Eds.), Readings in Artificial Intelligence and Software Engineering, Morgan Kaufmann, San Mateo, CA, 1986.

[20] C. Rich, R.C. Waters, The Programmer's Apprentice, Addison-Wesley, Reading, MA/ACM Press, Baltimore, MD, 1990.

[21] S.N. Roberts, R.L. Piazza, D.G. Katz, A portable assembler reverse engineering environment (PARE), in: Proc. 3rd Working Conference Reverse Engineering, Monterey, CA, 1996, pp. 77–85.

[22] E. Soloway, K. Ehrlich, Empirical studies of programming knowledge, IEEE Trans. Software Engineering 10 (5) (1984) 595–609. Reprinted in: C. Rich, R.C. Waters (Eds.), Readings in Artificial Intelligence and Software Engineering, Morgan Kaufmann, San Mateo, CA, 1986.

[23] G. Struble, Assembler Language Programming: The IBM System/360, Addison Wesley, Reading, MA, 1969.

[24] R.C. Waters, Program translation via abstraction and reimplementation, IEEE Trans. Software Engineering 14 (8) (1988) 1207–1228.

[25] L.M. Wills, Automated program recognition: A feasibility demonstration, Artificial Intelligence 45 (1–2) (1990) 113–172.

[26] L.M. Wills, Automated program recognition by graph parsing, Technical Report 1358, MIT Artificial Intelligence Lab., Cambridge, MA, 1992 (Ph.D. Thesis).

[27] L.M. Wills, Flexible control for program recognition, in: Proc. 1st Working Conference Reverse Engineering, IEEE Computer Society Press, Silver Springs, MD, 1993, pp. 134–143.

[28] L.M. Wills, Using attributed flow graph parsing to recognize clichés in programs, in: J. Cuny, H. Ehrig, G. Engels, G. Rozenberg (Eds.), Graph Grammars and Their Application to Computer Science, Lecture Notes in Computer Science, Vol. 1073, Springer, Berlin, 1996, pp. 170–184.