Journal of Applied Sciences

ISSN 1812-5654

science alert

ANSInet
an open access publisher
http://ansinet.com

Journal of Applied Sciences 10 (17): 1957-1962, 2010
ISSN 1812-5654

Program Translation via a New Dependence Model

Zine-Eddine Bouras and Amer Nizar AbuAli
Department of Software Engineering, Faculty of Information Technology,
P.O. Box 1 Philadelphia University-19392, Jordan

Abstract: Program translation takes a program written in some source language and creates an equivalent program in some target language. This process is very important in software maintenance, particularly when the source program is written in old language such as Cobol and Fortran. The goal of translation is efficiency and readability of the target program. In this study, we present a new approach to translate COBOL program via a dependence model in order to capture dependences between instructions. This model makes process of translation more adequate than the current approaches.

Key words: Program analysis, program translation, dependence relationships

INTRODUCTION

Maintainers of application software systems must often work with legacy systems. Legacy systems are systems that have evolved over many years and are considered irreplaceable, either because re-implementing their function is considered to be too expensive (Arranga *et al*., 2000). Because of their age, such systems are likely to have been implemented in a limited conventional language such as COBOL (Waters, 1988). Most Software managers agree that the only way to keep these systems competitive is to translate it in a new language where it can be more easily maintained and adapted to new business requirement (Chu, 1993; De Lucia *et al*., 1997). In this case, program is translated from language that may be in some way obsolete into another language where, it can be understood and then maintained. To resolve this problem some approaches has been proposed by Arranga and Price (2000), Boxer (1988), Van den Brand *et al*. (1997), Bruer *et al*. (1995), Chu (1993), Waters (1988) and Arranga *et al*. (2000). A large number translates program literally line-by-line. Others operate via abstraction and reimplementation, where abstraction gives a global analysis of the source program. Reimplementation step creates a program in target language from the abstraction description.

CURRENT APPROACHES

During the last decade several approaches have been proposed to resolve translation of program (Arranga and Price, 2000; Van den Brand *et al*., 1997; Bruer *et al*., 1995; Chu, 1993; Faust, 1981; Waters, 1988; Arranga *et al*., 2000). Among these, two relevant techniques have been used theoretically and practically. The first one called translation via transliteration and refinement (Boxer, 1988), transliterates the source program into the target language on a line-by-line basis by translating each line in isolation. Various refinements are then applied in order to improve the target program produced. This approach has many advantages. The most important is the use of divide-and-conquer strategy to satisfy the goal of translation. Another advantage is that the localised nature of transliteration step makes it easy to encode the basic knowledge needed for translation (Waters, 1988). Also it makes it easy to construct families of translators. However, transliteration can be blocked by the fact that the source languages may support primitive construct which are not supported by the target language (for example the use of GOTO). Also transliteration complicates refinement. Transliterations consider program as a set of line (but not instruction), indeed this approach doesn't preserve the semantics in the target program. Being implicit the semantics of programs can't be grasped by a tool that operates on textual translations (Bouras *et al*., 2000; Khammaci and Bouras, 2002).

The second one, named translation via abstraction and reimplementation, operates in two steps (Faust, 1981; Bruer *et al*., 1995). The abstraction step performs a global analysis of the source program in order to obtain an understanding representation to represent the essential semantic features. The reimplementation step takes the abstraction description produced by the first step and creates program in the target language that implements this description.

The most important advantage is that, while translation via transliteration and refinement is designed

Corresponding Author: Zine-Eddine Bouras, Department of Software Engineering, Faculty of Information Technology,
P.O. Box 1 Philadelphia University-19392, Jordan

to facilitate achieving the primary goal of translation (i.e., correctness), translation via abstraction and reimplementation is specifically designed to facilitate achieving other goal of translation such that there is no a priori reason for abstraction ever to be blocked since, the result of abstraction is not constrained by the target language. However, this approach has a fundamental problem of incompleteness. Another disadvantage of the abstraction and reimplementation is that it is more complicated than transliteration and refinement. Finally the target language is a subset of the source language and then it remains with the primary constraints (Carter *et al*., 1994).

We propose another approach, which capture all relations of dependence of the source program. Our approach combines the main advantages of current approaches to achieve the goal of translation and use a formal method to capture relation of dependence. Our abstraction step performs a thorough global analysis of the source program. The reimplementation step takes the abstract description and creates a primary program in the target language. In order to improve this primary program, an automatic refinement step is added.

PROGRAM MODELLING

In our approach, a Cobol program is modelled formally by on an internal form that capture flow dependencies in order to permit automatic data and control analysis. On other hand flow dependencies that are implicit will be explicit by this internal form.

Internal form: The internal form that we propose is due to the logical structure of program and programming concepts. At this step of our research ,we interested by a restricted COBOL programming language.
In the context of program comprehension and program translation, a dependence relationship is defined formally by the triplet:

<Ac, Ctr, As>

It means that target actions Ac and source actions As have a dependence relationship according to the constraint Ctr (Bendelloul *et al*., 1997; Bouras *et al*., 2000). Elementary actions Ac and As are formally defined by the Cartesian product:

Var x Act x IdDep

where, Var can be program, control, call variable or formal parameter.

Act can be definition, control or reference action. IdDep is an unique identifier corresponding to an instruction site, an effective parameter or to a formal parameter.

Ctr is defined by the triplet:

<Cdt, Exp, Sem>

Cdt is a condition, which must be true to execute actions Ac. It expresses the control constraint.

Exp is an expression to evaluate actions Ac. It expresses the data constraint.

Sem is a dependence relationship. It can be a Flow Dependence (FD), Control Dependence (CD), flow and control dependence (FCD), Formal Parameter Flow (FPF), Actual Parameter Flow (APF).

The internal form of a given program consists of modelling data and control structures.

MODELLING COBOL INSTRUCTIONS

To illustrate our approach, we use program DIV of Fig. 1:

Identification division:
00100 PROGRAM-ID <prg_name>.
The internal form of this instruction is:
< prg_name, DEC, i>, <Ø, prg_name, FC>, {<…>}.
It means that program declaration (prg_name) at the statement i depends on the program header by control flow FC.

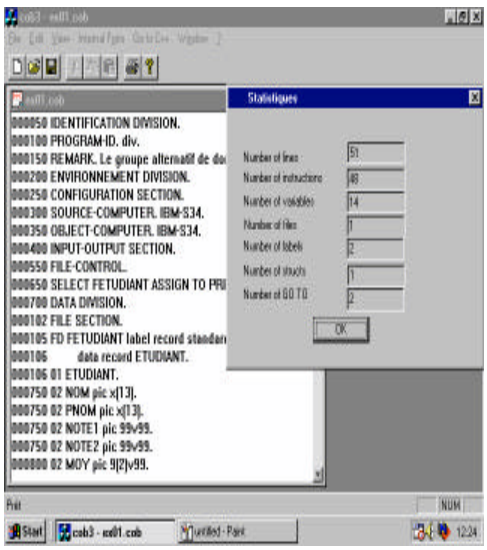

Fig. 1: DIV a given Cobol program

Data division:
FD<file_name> label record standard/omited
data record <name_records>
is formalized by:
<file_name, DEC, i> < Ø,char[31], FC> <prg_name, val, i>
For example, the internal form of:
FD fano label record standard data record EMPLOYE is
< fano, DEC, i> < Ø, FILE fano, FC> <DIV, val, i>,
< fano, DEC, i> < Ø, char[31], FC> <DIV, val, i>

Records:
num_level rec_name
Its internal form is:
<rec_name,DEC,i><Ø,STRUCTURE,FC><prg_name,val,i>
for example, the internal form of 000100 01 AGENT is:
<AGENT, DEC, i>< Ø, STRUCTURE, FC> <DIV, val, i>

PIC:
<var_name> PIC <type> has the following form :
<var_name,DEC,i>< Ø,type,FC><var_name_pred,val, i>
For example, the internal form of:

01 AGENT PIC 99
05 ECODE PIC 99
05 ENAME PIC X(10)
05 PMONTANT PIC 9(2)V9(3)
is:
{< agent, DEC, i>< Ø, structure, FC> < div, val, i>,
< ecode, DEC, i>< Ø, int, FC> < agent, val, i>
< ename, DEC, i>< Ø, char[10], FC> < agent, val, i>
< pmontant, DEC, i>< Ø, double, FC>, < agent, val, i>

Procedure division: To illustrate internal form of Procedure Division, we formalize instructions OPEN and ADD.

Open:
Internal form of
OPEN <INPUT/OUTPUT> <file_name> is:
<prg_name,IN/OUT,i><Ø,file_name=fopen(file_name,rt/wt), FD><prog_name,val,i>
for
OPEN INPUT fano
OPEN OUTPUT fsor we have:

< fano,IN,i><Ø, fano=fopen(fano,rt), FD>, <DIV,val,i>
<fsor,OUT,i><Ø,fsor=fopen(fsor,wt)FD><DIV,val,i>

Add: Add has two forms:

form1 ADD <var1> <var2>…<varn> TO <varm>
form2 ADD <var1> <var2>…<varn> GIVING <varm>

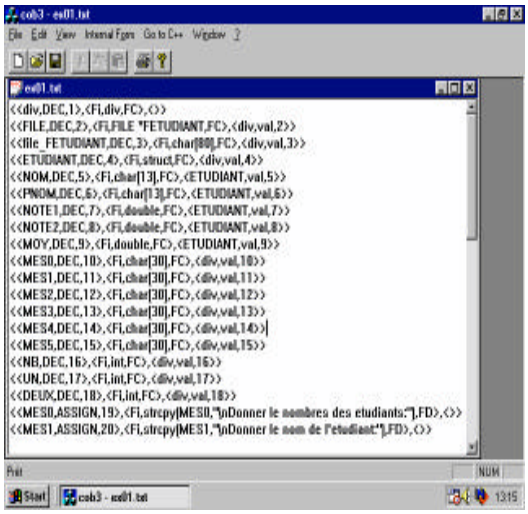

Fig. 2: Internal form of program DIV

Their internal forms are:

- <varm,ASSIGN,i><Ø,varm=varm+var1+…+varn,FD>, {<var1, val, i>,…<varn, val, i>}
- <varm,ASSIGN,i><Ø,varm=var1+var2+…+varn, FD>, {<var1, val, i>, …<varn, val, i>}

Other instructions are presented in our previous work (Bendelloul *et al*., 1977; Khammaci *et al*., 2000).

Every program is seen as a set of triplets. Being formal this modeling allows automatic translation. Fig. 2 shows the internal form of program DIV.

THE TARGET LANGUAGE

Target language must solve some problems where the most important are:

- **Record:** Cobol variables are declared under tree form.
- **Copy:** Cobol permit to copy two records having the same size but different fields
- **Goto:** There is not any concept of level, we can perform GOTO from any level to another level.
- **Perform:** Cobol permit to perform a part of a given program between two labels without specifying it as procedure or subroutine

From this we justify the choice of target language.

The Data Base Management System Languages (DBMS) are adequate to solve problems i and ii but not iii and iv.

Java is a modern language but it has some problems with translation from COBOL. The concept of record

doesn't exist in Java; to declare a variable with 49 levels we must declare 49 classes. Copying two records having the same sizes but having different fields is prohibited in Java. The instruction Goto doesn't exist in Java (Hamilton, 1996). Then Java is not adequate as target language to translate a program COBOL.

Copying two records having the same sizes but different fields is prohibited in Pascal (Delannoy, 1994). The problem with this instruction hasn't any solution. Also Pascal is not adequate as target language to translate a program COBOL.

Pascal is a sub language of Delphi. Delphi inherits its problems, expect the notion of object oriented programming that solves the problems of Goto and Perform. Problems with Copy are not solved (Engo, 1997). As Pascal and Java Delphi is not adequate.

With C language notion of Record exists (Struct) and problem to declare variable in the form of tree structure is solved. Copying two records having the same sizes but having different fields is possible with C. Goto exists but is not possible between two levels, same problem with Perform (Young, 1993).

C++ has the same features as C, in addition the notion of object-oriented programming solves the problems of Goto and Perform (Young, 1993). Then C++ solves the four problems of Cobol. It is an adequate language to translate Cobol programs. We adopt C++ as target language to translate Cobol programs.

FROM INTERNAL FORM TO C++

Translation of internal form to C++ needs to create two files: the header file and the source code file.

Header file:

<rec_name,DEC,I><Ø,STRUCTURE,FC><prog_name, val,i>
is represented in the header file by:
typedef < record_name>
{data} < record_name>

Others instructions are translated with the same process. For example to translate
{< agent, DEC, i>< Ø, structure, FC> < div, val, i>},
{< ecode, DEC, i>< Ø, int, FC> < agent, val, i>},
{< ename, DEC, i>< Ø, char[10], FC> < agent, val, i>},
{< pmont, DEC, i>< Ø, double, FC> < agent, val, i>}.

We obtain:
typedef agent
{int ecode;
ename char[10],
Double pmont;
} agent;

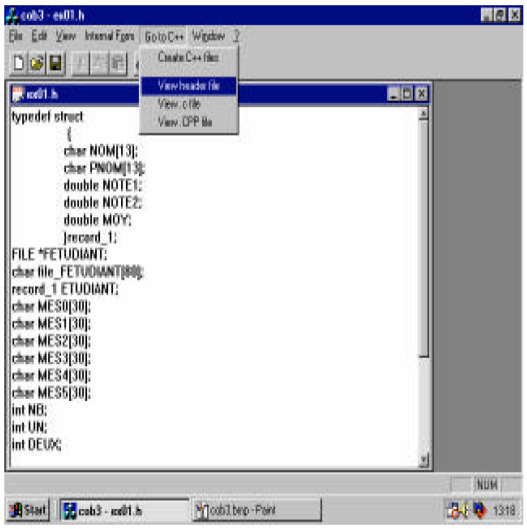

Fig. 3: The header file of program DIV

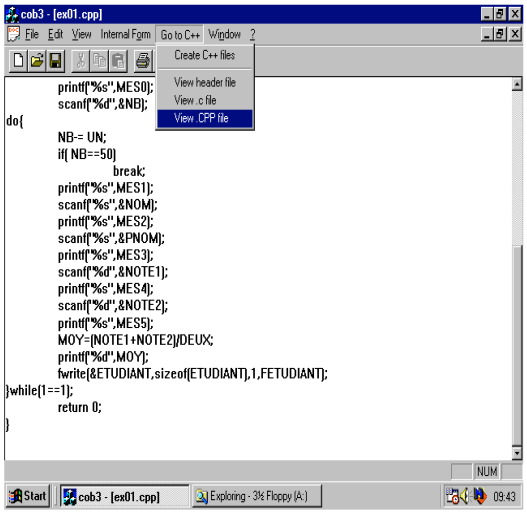

Fig. 4: DIV in C++

In Fig. 3 we present the print screen of the header file of DIV.

Source code file (.cpp): To create file .cpp we call the following modules:

#includestdio.h,
#includedos.h,
#includestdlib.h,
#includestring.h,

with the previous header file. In Fig. 4 we have the screen of C++ program of DIV.

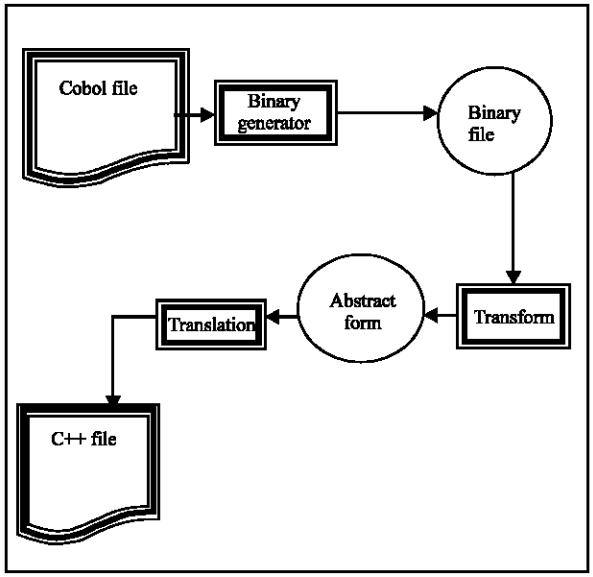

Fig. 5: TRANSCOBC++ architecture

Finally we can obtain a complete translated program in C++ with the same modelling.

TRANSCOBC++

We have designed and implemented a prototype based on our approach named TRANSCOBC++ (TRANSlation from COBol to C++). It has been developed under Visual C++ 4.0 environment and is formed by three modules: COB2iff, iff2txt and Translator (Fig. 5).

In the first step, Cob2iff, which is a binary generator, makes lexical, syntactic and semantic analysis of a given Cobol source code. It gives table of variable definitions and references and table of call points. From these tables, it elaborates program internal form and instrumented source code. Then, iff2txt converts the binary file to textual file and displays the internal form. Finally Translator translates internal form to C++ programs. We have also included a possibility to refine automatically and manually C++ program. C++ program obtained can be carried out.

CONCLUSION AND FUTURE WORKS

Traditional program translation takes a program written in some source language and creates a semantically equivalent program in some target language. In this study, we have described the current approaches of program translation and their insufficiencies.

A translation via transliteration and refinement is the major approach, in which the source program is first transliterated into the target language and a line-by-line basis and various refinements are then applied to improve the produced target program.

In many cases, it serves the purpose of correctness but it is quite limited to satisfy the other goals, such as the improvement of readability, maintainability and reusability. Another approach, translation via abstraction and reimplementation, has been proposed to satisfy these goals. However, this approach is not able to be applied to programs of commercial size and complexity.

We have proposed a new solution that overcomes these insufficiencies. Our approaches use the concept of dependence model, which capture all relations of dependence of the source program.

Our approach need to see the results of translating large samples of Cobol programs into C++ and to compare both their static and dynamic behavior for both Cobol and the resulted C++ programs. In such case only we can infer the affectivity of the translation. That is our future work and of course we will change these programs to software which will be suitable, like Visual Studio 2008/2010 or Java.

In future work of course, we will change these programs to software which will be suitable to future like Visual Studio 2008/2010.

REFERENCES

Arranga, E., I. Archbell, J. Bradley, P. Coker, R. Langer, C. Townsend and M. Weathley, 2000. In cobol's defense. IEEE Software, 17: 70-72.

Arranga, E.C. and W. Price, 2000. Fresh from Y2K: Whats next for COBOL. IEEE Software, 17: 16-20.

Bendelloul, M.S., Z.E. Bouras, S. Ghoul and T. Khammaci, 1997. Assistance a la compréhension de programmes: Un modèle et un algorithme de fragmentation. Genie Logiciel, 45: 32-42.

Bouras, Z.E., S. Ghoul and T. Khammaci, 2000. A new approach for program integration. South Afr. Comput. J., 25: 3-11.

Boxer, R.K., 1988. A translation from structured FORTRAN to Jovial/j73. IEEE. Trans. Software Eng., 14: 1207-1228.

Bruer, P.T., K. Lano and H. Haughton, 1995. Reverse engineering cobol via formal methods. Comput. J., 28: 47-56.

Carter, L., J. Ferrante and V. Bala, 1994. XDP: A compiler intermediate language extension for the representation and optimization of data movement. Int. J. Parallel Programming, 22: 485-518.

Chu, W.C.A., 1993. Re-engineering approach to program translation. Proceedings of the International Conference on Software Maintenance, Se. 27-30, IEEE Computer Society Press, pp: 42-50.

De Lucia, A., G.A. Di Lucca, A.R. Fasolino, P. Guerra and S. Petruzzelli, 1997. Migrating legacy systems to wards object-oriented platforms. Proceedings of the International Conference on Software Maintenance, (ICSM'97), Bary, Italy, pp: 122-129.

Delannoy, C., 1994. Programmer en Turbo Pascal 7.0. ED. Eyrolles, France.

Engo, F., 1997. How to Program Delphi. Ziff-Davis Press, New York.

Faust, G.G., 1981. Semi automatic translation of cobol into hibol. Master Thesis, MIT Cambridge MIT/LCS/TR-256.

Hamilton, M.A., 1996. Java and the shift to net-centric computing. IEEE Comput., 29: 31-39.

Khammaci, T. and Z.E. Bouras, 2002. Versions of program integration. Handbook Software Eng. Knowledge Eng., World Scientific, 2: 495-516.

Khammaci, T., Z.E. Bouras and M.S. Bendelloul, 2000. Program understanding assistance: A role-based decomposition. Proceedings of the 12th International Conference on Software Engineering and Knowledge Engineering, SEKE2000, 6-8 July, Chicago, USA., pp: 336-343.

Van den Brand, M.G.J., M.P.A. Sellink and C. Verhoef, 1997. Obtaining a COBOL grammar for legacy code for reengineering purposes. Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specification, 1997. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.25.8267.

Waters, R.C., 1988. Program translation via abstraction and reimplementation. IEEE Trans. Software Eng., 14: 1207-1228.

Young, M.J., 1993. Visual C++ for Windows. Sybex, Inc., France.