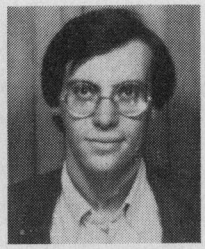
architecture," in *Proc. 10th IFIP Conf. Syst. Modeling and Optimiz.*, New York, Aug. 31-Sept. 4, 1981, R. F. Drenick and F. Kozin, Eds. Springer-Verlag, 1982.
[4] P. J. Courtois, *Decomposability: Queueing and Computer System Applications*. ACM Monograph Series, 1977.
[5] F. Baskett, K. Chandy, R. Muntz, and F. Palacios, "Open, closed and mixed networks of Queues with different classes of customers," *J. Ass. Comput. Mach.*, vol. 22, 1975.
[6] E. Gelenbe, "On approximate computer system models," *J. Ass. Comput. Mach.*, vol. 22, Apr. 1975.
[7] M. Eisenberg, "Queues with periodic service and changeover time," *Oper. Res.*, vol. 20, Mar./Apr. 1972.
[8] A. G. Konheim, "Service epochs in a loop system," presented at the *Symp. Comput. Commun. Networks and Teletraffic*, Polytechnic Inst. Brooklyn, Apr. 4-6, 1972.
[9] G. B. Swartz, "Analysis of a SCAN service policy in a gated loop system," Monmouth College, West Long Branch, NJ.
[10] A. Chang and S. S. Lavenberg, "Work-rates in closed queueing networks with general independent servers," *Oper. Res.*, vol. 22, 1974.
[11] Y. Bard, "An analytic model of the VM/370 system," *IBM J. Res. Develop.*, vol. 22, Sept. 1978.
[12] M. Becker and R. Fortet, "Projector method and iterative method to solve a packet switching network node. Validation by simulation," in *Measuring, Modelling and Evaluating Computer Systems*, H. Beilner and E. Gelenbe, Eds. North-Holland, 1977.
[13] L. Kleinrock, *Queueing Systems, Vol. 2: Computer Applications*. New York: Wiley-Interscience, 1976.
[14] D. P. Gaver, "Analysis of remote terminal backlogs under heavy demand conditions," *J. Ass. Comput. Mach.*, vol. 18, 1971.
[15] J. Zahorjan, N. P. Hume, and C. Sevcik, "A queueing model of a rotational position sensing disk system," *Infor.*, vol. 10, 1978.
[16] A. G. Konheim and B. Meister, "Service in a loop system," *J. Ass. Comput. Mach.*, vol. 19, 1972.
[17] P. J. Kuehn, "Multiqueue systems with nonexhaustive cyclic service," *Bell Syst. Tech. J.*, Mar. 1979.



Jean René Ménand was born in Albert, France, on March 16, 1954. He received the engineering degree from the Institut Supérieur d'Electronique du Nord in 1977, and the degree of Docteur Ingénieur from the University of Paris VI, France, in 1980.

His research interest includes distributed systems, multiprocessing architecture, and performance evaluation. He has participated in the implementation of the multiprocessor architecture which is modeled in this paper. Presently, he is with the Central Research Laboratory, Thomson CSF, Orsay, France, where he participates in the development of the EXEL language implementation on a personnal computer.

Monique Becker was born in France in 1945. She graduated from Ecole Normale Supérieure de Jeunes Filles in 1968, passed the mathematics "agrégation," and received the State Doctorate degree in 1976.

She joined the National Center of Scientific Research. She directed the thesis of J. R. Ménand. She has the responsibility for a group of researchers working on performance evaluation. A part of this group is in Paris and belongs to ERA 592, a part of this group is in Grenoble and belongs to ERA 534.

Short Notes

A System to Automatically Analyze Assembled Programs

V. HAYWARD AND A. OSORIO

Abstract—**An original system to perform an automatic analysis of assembled programs is presented. Executable programs are analyzed from the description of the machine on which they run and are translated into an intermediate language taking into account the particularities of the considered machine. The system was primarily designed as the first step of a project for transferring programs from one machine to another. The final goal of the project is to achieve an even utilization of computer resources for a real-time controlled robot, on the basis of partially dedicated processors. At the present time, the actual implementation provides a tool for studying the theoretical aspect of machine-level program analysis. Nevertheless, other applications can be found in program debugging and assembled program validation.**

Index Terms—**Machine description, machine program analysis, program debugging, program transfer, real-time robot control, resources optimization.**

I. Introduction

The control of a robot by means of a computer often requires unusual instantaneous processing needs. This fact becomes more evident with the advent of advanced robots equipped with complex sensory systems. New task programming techniques also tend to reduce the planning phase of the task execution in order to allow a more flexible decision making at run time [9], [10]. The problem is usually solved by using very powerful machines, with a poor utilization of computer resources as a result.

Hence appeared the necessity of searching for some alternative solutions. Our research investigates a new approach whose underlying idea is to equalize processors' activities by distributing the load among different processors of a system, given that all of them may not be overloaded at the same time. The optimization process should be performed in real time, according to the situation at hand. Consequently, we must be able to transfer executable code from one machine to another.

The work described here presents the theoretical aspect of program analysis at a machine level. Such a technique will allow us to pick up at run time code segments in order to reprogram them in other processors.

The system automatically produces a representation of a machine program expressed at the register transfer level.

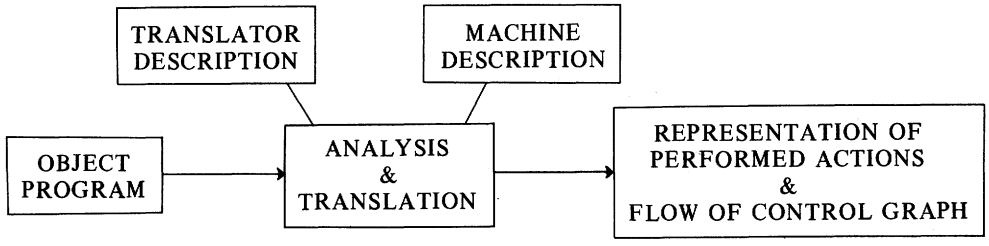
Given a description of the machine and the entry point of the program, one obtains a translation of the program in an intermediate language and a representation of the flow of control graph.

As a result, this system may be used in applications that require a precise knowledge of the behavior of an assembled program.

An interactive version is now implemented and is already in use for program debugging.

Manuscript received March 1, 1982; revised July 1, 1982.
The authors are with L.I.M.S.I., 91406 Orsay Cedex, France.

0098-5589/83/0300-0210$01.00 © 1983 IEEE



A machine description is used by the translator acting at the machine instruction level. The analyzer, which depends only on the structure of generated programs, uses general algorithms.

II. The Translator

The main functions of the translator are to recognize a bit-string as one of the machine's instructions and then to give a symbolic representation of it, expressing the performed actions. These operations require a description of the machine.

A. Translator Source Language

Input to the translator is a continuous bit-string containing program data and instructions. The syntax of the instructions themselves is rather simple, although it may be quite variable even for a given instruction set. Each instruction is considered as a chain of variable length fields. The length and the meaning of each field may not be fixed and may depend on the value of other fields.

B. Translator Object Language

The translator produces programs in an intermediate language made of expressions of the following form:

TRANSFER($r(a)$, EXP(0 , $R(A)$))

where the "TRANSFER" operator assigns the value of the "EXP" expression to the $r(a)$ register. The "EXP" expression is built around operators which belong to "0," the set of the machine's operators. The operands "$R(A)$," are taken among the machine's registers. Notice that certain instruction fields may be considered as registers.

The registers are referred by the addresses "a" and "A," which, in turn, may appear as expressions. In the case of an implicit addressing, they are omitted.

Conditional actions are expressed as

COND(EXP(0′ , rb), TRANSFER(- - -))

where "EXP" is a Boolean expression.

C. Machine Description

Machine description systems are often complex tools involving a number of subsystems as parsers. The effective use of data can be achieved only after a considerable amount of processing [2], [3]. We make use of a notation which is close to the internal representation of data; thus the information is easily available to the application programs. The description is given under a declarative form.

1) Data Type and Operators: Machines are usually built around a number of data types and operators. Two data types are taken into account by the system. A bit-string can either be interpreted as a signed number (called "*SN*") or as an unsigned number ("*UN*") that can also be called an address. A length is associated with each data type.

Specific operators are defined by primitive operators which are supposed to have an infinite precision and by associating a specific data type to each of its operands. For instance, we may define the operators that compute an address by adding a 16 bit length base to an 8 bit signed offset as (+(*UN* 16, *SN* 8)).

The type of the result is defined by the operator that uses it. Here, it will be a (*UN* 16), considering that the memory referencing operator uses a (*UN* 16) quantity.

2) Registers: Registers are classified by their addressing mode: implicitly addressed registers are given a name and addressing is described for the explicitly addressed registers. The program counter plays a special role since its content has a predetermined meaning.

Some registers may have the double property of being implicitly or explicitly addressed. For example, the members of a general purpose processor register set are often referenced by an instruction field. In that case, although these registers have the property of explicit addressing, they can be considered as implicitly addressed, since they are readily known at the instruction analysis time.

Logical layout is described by subregisters defined on previously declared registers. It is also possible to describe logical registers by grouping.

Example: A description of the 8080 processor registers is as follows:

```
(E:MEM 8 0 65535)
(E:REG 1 0 7 (I:B 8, I:C 8, I:D 8, I:E 8, I:H 8,
              I:L 8, , I:ACC 8) )
(E:SS_DD_PAIRS 4 0 1 (I:BC 2(I:B 8, I:C 8)
                      I:DE 2(I:D 8, I:E 8)
                      I:HL 2(I:H 8, I:L 8) I:SP 16) )
```

where in the < *M i f* 1 > quadruple:

M is the mnemonic name, the prefix '*E*: ' or '*I*: ' indicates the addressing mode type;

f is the first address;

1 is the last address;

i is the increment of address or the size.

3) Instructions: The physical structure of the instructions is described by a context-free grammar whose terminal elements are instruction fields. They are defined by a length, a value, or a range of values. The length of an instruction is given by the sum of the lengths of its fields. The instruction fields can be named and referenced by their names.

D. Translator Generation

A template library is used to produce the appropriate translator for a given machine. Each template is a tree representation of a basic action. A template may represent a whole instruction or part of it, as an addressing mechanism or an indicator setting mechanism. A template includes subtemplates which can be conditionally activated according to the result of an instruction analysis. It should be noted that it is possible to build various template libraries for different applications. For instance, the flow of control determination can be achieved while only translating the instructions that affect the sequencing. It is also possible to construct a template library which produces an assembly-language-like notation.

E. Translator Operation

Each encountered machine code instruction is parsed into a syntax tree according to the rules of the instruction set gram-

mar. Each leaf of the syntax tree points to an instruction field. The grammar must be constructed in such a manner that the syntax tree contains a marked node. This node points to an entry in template library. The corresponding template is recursively evaluated by the activation of its subtemplates.

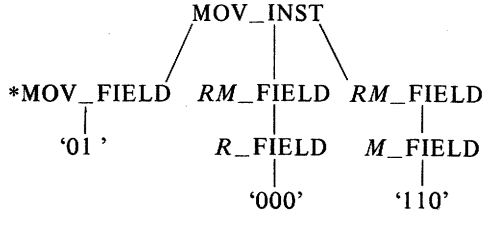
Since the activation of these subtemplate may depend on the presence of given nodes in the syntax tree, the translation process can be controlled for each version of an instruction. This feature avoids having too many categories.

An example is given below for the MOV *r*, *M* instruction of the 8080 processor.

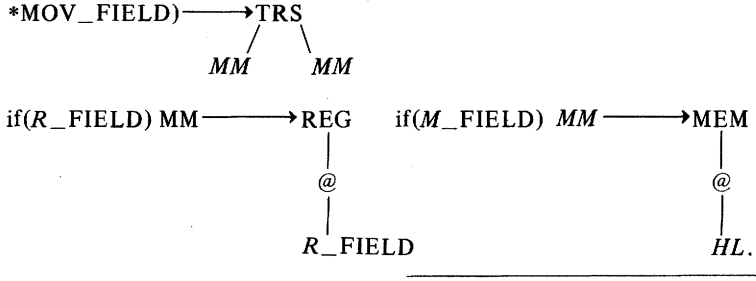
Physical Description:

MOV_INST : : = *MOV_FIELD(val=1, len=2) *RM*_FIELD *RM*_FIELD
*RM*_FIELD : : = *R*_FIELD(0<val<5, len=3) | *A*_FIELD(val=7, len=3) |
*M*_FIELD(val=6, len=3).

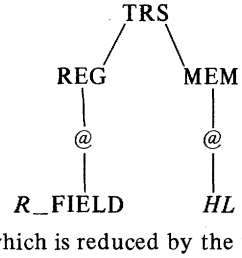
Input to the translator is '01000110. . . . . '
We obtain the following tree:



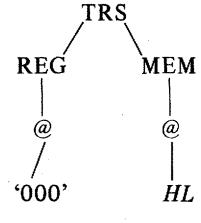
In the template library we have



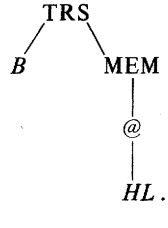
We finally obtain by replacement



which is reduced by the translator to



and then to



III. Flow of Control Determination

It is necessary to determine the flow of control graph when analyzing an assembled program, for the physical location in memory of the program segments cannot be known as one could do by parsing the text of a high-level language program, nor can the analysis be performed by a sequential scanning of the memory content. The determination of the flow of control graph is a by-product of such an analysis.

A. Representation

The representation is achieved using a block of code model.

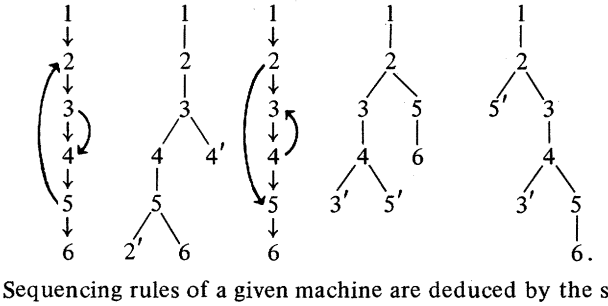
A block of code is defined as a set of instructions containing no label, ended or not by a branch instruction [1], [5].

The flow of control graph is an oriented graph whose arcs represent block of code and whose vertices contain labels (in this case, addresses).

B. Algorithm

The algorithm makes use of the fact that is always possible to define a free tree on a flow of control graph [6].

The graph is represented by a tree; one of its leaves contains the exit address of the program; the other leaves contain addresses referenced at least once in the nodes of the tree.



Sequencing rules of a given machine are deduced by the system from the description of the branch instructions. A marking algorithm builds a block of code table and the tree representation of the flow of control.

One can shortly express this algorithm using a Pascal-like procedure:

```
PROCEDURE flow_of_control( address ) ;
  BEGIN
    IF sequential instruction AND NOT marked THEN
      BEGIN
        create a block ;
        start of the block := address ;
        WHILE sequential instruction AND NOT
              marked DO
          BEGIN
            mark instruction ;
            next instruction ;
            end of the block := address
      END ;
    IF branch instruction THEN
      BEGIN
        add a left leave to the tree ;
        flow_of_control( branching address ) ;
        IF conditional instruction THEN
          BEGIN
            add a right leave ;
            flow_of_control ( next instruction address)
          END
      END
    IF marked instruction AND address <> start of
          block THEN
      BEGIN
        split the block in two blocks ;
        update tree with a marked leave
      END
END.
```

C. Limitations

The full generality of the translator is impaired by certain coding techniques that can be used at the machine level. For instance, it is not possible to deal with programs which modify themselves. The system suffers other limitations, especially when addresses cannot be easily deductible, as in indexed branching techniques.

IV. Transformations

The first phase of the processing gives a symbolic representation of all the actions which are performed by the analyzed program. We obtain, under a tree form, a translation of all operations and assignments performed on the machine's registers.

Further processing depends on the desired application. In the case of machine code transcription, it is not necessary to perform a complete analysis of the data flow. Such an analysis requires many computations and is performed by algorithms whose execution times increase rapidly with the size of the problem.

Nevertheless, it is useful to apply transformations. The *Ti* transformation is described by Aho and Ullman [1] in the optimization of code sequences without jumps. We apply the *Ti* transformation for discarding useless register assignments which appear in a crude translation of machine instructions. A machine program often uses only part of the actions actually performed (indicators settings, autoincrements).

An assignment is considered useless if the affected register is not referenced in the interval ended by its next assignment. Here, in the case of an explicit addressed register, the transformation can occur only when its address is known by the instruction analysis.

V. Conclusion

This work demonstrated the feasibility of machine code analysis, taking into account a sufficiently broad class of machine architectures. Nonetheless, inevitable limitations restrict the class of the analyzed programs. These limitations can be avoided in two ways. First, assumptions can be made about the structure of the programs themselves. These assumptions can be included in the knowledge that the system has about the analyzed programs. Second, modern compilers allow us to easily modify their code generation phase. It is therefore possible to obtain compilers so that the code they produce can be analyzed without ambiguities. Furthermore, the approach use in this work allows immediate applications, such as program debugging, and provides a useful tool for future research in the field of assembled programs analysis.

References

[1] A. H. Aho and J. D. Ullman, *The Theory of Parsing, Translation and Compiling, Vol. II: Compiling*. Englewood Cliffs, NJ: Prentice-Hall, 1973.

[2] M. R. Barbacci, G. E. Barnes, R. G. Cattel, and D. D. Seiwiorek, "The ISPS computer description language," CMU Tech. Rep., 1981.

[3] V. Chu, "An algol like computer design language," vol. 8, pp. 607–617, Oct. 1965.

[4] V. Hayward, "Langages d'analyse de programmes assemblés pour différentes machines et applications de ces langages," thèse D. I., Université de Paris-Sud, Centre d'Orsay, France, Nov. 1981.

[5] K. Kenedy, "A survey of compiler optimization techniques," Le point sur la compilation, Cours des communautés européennes, Jan. 1978.

[6] D. E. Knuth, *The Art of Computer Programming, Vol. I: Fundamental Algorithms*. Reading, MA: Addison-Wesley, 1973.

[7] K. Ripken, "A formal method for describing machines code generation with local optimizations," Le point sur la compilation, Cours des communautés européennes, Jan. 1978.

[8] R. C. Waters, "Automatic analysis of the logical structure of programs," M.I.T., Cambridge, MA, Tech. Rep., Dec. 1978.

[9] S. Mujahu and R. Goldman, "AL user's manual," Stanford Artificial Intell. Lab., Stanford, CA, Memo. 323, Jan. 1979.

[10] R. Paul, "Manipulator language."