LAURA, A System to Debug Student Programs

Anne Adam and Jean-Pierre Laurent
University of Caen, 14032—Caen Cedex, France

Recommended by Patrick J. Hayes

ABSTRACT

An effort to automate the debugging of real programs is presented. We discuss possible choices in conceiving a debugging system. In order to detect all the semantic errors, it must have a knowledge of what the program is intended to achieve. Strategies and results are very dependent on the way of giving this knowledge. In the LAURA *system that we have designed, the program's task is given by means of a 'program model'. Automatic debugging is then viewed as a comparison of programs. The main characteristics of* LAURA *are the representation of programs by graphs, which gets rid of many syntactical variations, the use of program transformations, realized on the graphs, and its heuristic strategy to identify step by step the elements of the graphs. It has been tested with about a hundred programs written by students to solve eight different problems in various fields. It is able to recognize correct programs even if their structures are very different from the structure of the program model. It is also able to express exact diagnostics of errors, or at least to localize them. It could be an effective tool for students programmers.*

0. Introduction

Programming is now the subject of numerous studies. Methodology of Programming, Optimization, Synthesis of Programs, Automatic Verification and Debugging, Transformations of Programs are existing fields of research, the common goal of which is to make the production of correct and efficient programs easier.

In order to write correct programs, attempts have been made to impose constraints on their structure. The works on Structured Programming [5, 14, 19, 27] are the first basis of a real methodology of Programming. 'Well structured' programs become objects easier to deal with. In particular it will be easier to prove that they are correct, the ideal method being to write and prove a program simultaneously. The use of recursive languages may be an efficient way to achieve this goal. As a matter of fact, if enough recursivity and calls of procedures are used, it is possible to write programs without loops. Their structure is more simple and particular techniques may be used for their verification [9, 10].

We are concerned with a slightly different problem, that is to debug programs that have already been written. If an iterative language has been used, these programs may contain loops and have a complex structure. Then, the debugging needs other methods. Henceforward, by Automatic Verification of Programs we understand the following problem: *to establish whether a given program is correct or not, and if not, to obtain enough elements of information so that the necessary rectifications can be made.* It is a difficult task, especially since the grossest errors are sometimes the most difficult to find. The implementation of a *correct algorithm*, well appropriated to the problem that must be solved, often gives rise to an *incorrect program.* Using a good compiler, it is easy to eliminate the errors concerning the syntax of the programming language used. But often, the obtained program, though *syntactically correct*, still contains *semantic errors.* These alter the meaning of the program and prevent it from giving the expected results.

Various methods for automatic verification have been considered ([6, 12, 16, 18, 23, 24, 33, 41] etc.). Some have a great theoretical interest but they often need very heavy proofs of theorems and thus are difficult to automate. So, few of these methods have been put into practice. Moreover, these methods are especially useful to prove the correctness of a program. If it is incorrect, they seldom give information about the nature of the errors. In this respect, the most interesting work, of which the main goal is to discover errors, is probably the system of Ruth [39]. But its performances are also limited, we shall see why later.

The goal of this paper is to present the LAURA system, that we have built up in order to pin-point the semantic errors, or at least to localize them. It has been tested with real programs written by student programmers on various subjects. It recognized those that were correct and expressed diagnostics of errors in the others.

In order to debug a program it is absolutely necessary to have information on what it is intended to achieve. This information is given to our system by means of an implementation of the algorithm to be used. This implementation is supposed to be correct and we call it the *program model.* The program that the system has to debug is an *implementation of the same algorithm*, syntactically correct, but which may contain *semantic errors.* The system will compare the two implementations.

In this context, the ability to apply *program transformations*, either systematically or according to heuristics, is a determining factor. It makes it possible to recognize that a certain part of one program calculates the same functions as a certain part of the other. These parts then can be identified. If a total identification is possible, the student program must be declared correct. Otherwise, the errors are limited to the unidentified parts. The system may recognize an error, express a diagnostic and correct the errors by itself in order to carry on with identifications.

But let us point out that a difference may reveal an error or only a variation. The interpretation of differences is very difficult. It needs not only an exact knowledge of the problem the program has to solve but also a thorough knowledge of the field in which the problem has a meaning.

The LAURA system may determine that two programs calculate the same functions, which means that they will produce the same results with round-off errors (this is a particular form of equivalence). Thus our work also concerns the problem of automatic checking for the equivalence of two programs. It also makes obvious the great power of program transformations due to their application and especially their activation by an automatic system. This is only possible if the goal is clearly expressed. This is the case for LAURA since it considers debugging from the angle of a comparison between two programs, the model and the candidate.

1. What is useful for debugging?

This section attempts to abstract the ideas that led us in conceiving our system of debugging. Other possibilities are mentioned and discussed. The solutions selected by LAURA are explained.

1.1. Knowledge and debugging

We examine which knowledge a system may use in order to determine if a given program contains semantic errors.

1.1.1. *A priori detection of errors*

In some cases, it is possible to detect errors in a program without any knowledge of its intentions. These errors concern the dynamical logic of programs and are independent of the particular problem to be solved.

For example, it is possible to recognize that one variable is used which is not always defined, or on the contrary to recognize that one definition of a variable is never used. Also, it is possible to discover that no instruction in a loop defines a new value for any variable used in an output test....

These kinds of errors were studied in FORTRAN programs by Flavigny [17], who called them 'anomalies' and later in LISP programs by Wertz [43] who called them 'inconsistencies'. The methods that they used are not suitable if the system must find all the semantic errors in a program. Many are too tightly related to the task the program has to perform. The program may be free of 'anomalies' without performing the task at all.

If we hope that a system will be able to determine whether or not a program executes the expected work, and to discover each semantic error, it is absolutely necessary to give it *a certain knowledge of the goal.*

Yet, searching for a priori errors with simple and direct methods may be a first and efficient step when debugging programs. When it constructs the graph (see Section 1.2) and when it applies systematic transformations (see Section 2.1) LAURA can easily detect many errors of this kind.

1.1.2. *Static description of the program's task*

The description of the program's task may consist of sets of assertions. The final results expected from the program are expressed by means of one set of output assertions. One set of input assertions corresponds to the data properties. Then it must be proven that the program holds through from input assertions to output assertions. Floyd [18], Naur [36] and Hoare [22], were the first to consider and to formalize this method that has since been developed by many researchers. We shall no longer describe the methods of program verification that use assertions. They are now classic in Artificial Intelligence. We are only going to discuss the main objections that may be made to these methods and that have led us to try another approach.

(1) Firstly, it is very difficult to give complete sets of assertions that describe the problem entirely. Let us consider for example the problem: "find the greatest element of an array A of N numbers". In order to describe the goal to achieve, Waldinger and Levitt [41] propose the following set of output assertions:

$$\begin{cases} \text{MAX} = A(\text{LOC}) \\ A(0) \leq \text{MAX}, A(1) \leq \text{MAX}, \ldots, A(N) \leq \text{MAX} \\ 0 \leq \text{LOC} \leq N \end{cases}$$

They use this set in order to prove the correctness of their program. Yet, we can easily imagine programs that satisfy the same set of output assertions, even though the required task has not been performed at all. Such is the following program:

$$A(0) = 0; \quad A(1) = 1; \quad \ldots; \quad A(N) = N; \quad \text{MAX} = N; \quad \text{LOC} = N;$$

The suggested set of output assertions is thus insufficient. In addition, it is necessary to give assertions which state that each value in the initial array must still be in the final array, with the same number of occurrences.... This example of a very trivial problem illustrates how difficult it is to construct suitable sets of assertions to describe a program's task.

(2) Secondly, the methods based on assertions need many complex proofs to get invariants and to establish that a sequence holds through from one invariant to another.

In order to prove the correctness of the whole program, it must be cut into sections which are simple paths. At each extremity, it needs to have one set of internal assertions. In particular, invariants of loops are necessary. The formulation of invariants may be confided to the programmer. This would be paradoxical, because the formulation of suitable internal assertions quickly becomes a more difficult problem than writing a correct program. As for the automatic generation of invariants, several heuristic methods have been considered, for example by Katz and Manna [25], but they are still far from being totally efficient. The principal difficulties are that the system must generate a reasonable number of candidate formulas, and that the proofs that they are really invariants are generally complicated.

It is the same thing in proving that the instructions of a simple path lead to the extremity assertions from the origin assertions. It is usual to find papers where a long and very laborious proof is given manually to establish the correctness of a small program. For instance, in a paper called "Program proving without tears" (?) Ashcroft [6] establishes the correctness of one program of ten instructions. The proof needs six steps and one of them is itself made of twenty four steps....

It is doubtful that all these proofs can be automated, at least considering the actual power of formal calculus programs (MACSYMA [32], REDUCE [21]) and of automatic theorem provers [7, 8, 31, 37]. It would be all the more difficult since the system would be used to deal with real programs of medium size, for instance fifty instructions.

(3) Last but not least, the methods of program verification which are based on assertions are only used to determine whether the program is correct or incorrect. They give a boolean answer and if the program is incorrect, they do not make any constructive criticism to show what errors have been made. There is the same deficiency if the system does not succeed in proving the correctness of the program nor its incorrectness.

For all the reasons just put forward we think that assertions are not the most adequate and useful knowledge that a system must have in order to debug a program.

1.1.3. *Dynamic description of the program's task*

There is another way of giving the knowledge of the program's task to the system. Instead of describing *what* the program has to achieve one may give information about *how* it must proceed. In other terms one may give the system the *algorithm* needed to attain the goals rather than the goals themselves.

In order to give the algorithm, it is possible to use a specific language of algorithm description, such as the language 'lucid' used by Ashcroft [6] or the language used by Ruth [39]. The system of Ruth has to debug a student program which is written in a language without labels and appears as a list of actions. A Program Generation Model (PGM) is also given to the system, that also appears as a list of actions. From each action of the PGM, the system of Ruth may generate actions in the student program language, considering many possible syntactical choices. If the student program is derivable from the PGM it is assumed to be correct. Otherwise if an action cannot be derived, a diagnostic of error is made. In the system of Ruth, debugging is viewed from the angle of Program Synthesis.

We have selected another way to give the algorithm to the LAURA system: an implementation (in any classic iterative language) is given, which is assumed to be correct. From this program, the system constructs an internal representation of the corresponding calculus process (see Section 1.2). The representation of the calculus process implied by the student program is also constructed. So, *LAURA considers debugging from the angle of the comparison of two calculus processes.* Exact diagnostics can then be made in considering the non-recoverable differences.

The system of Ruth and the LAURA system have detected many errors in real programs. Both know the algorithm to be implemented and it is a determining factor to pin-point errors when the program is incorrect.

1.2. Representation and debugging

As we try to debug a program by comparing it with a program model, it is desirable to have a representation of programs, which is as independent as possible of the syntactic choices made by the programmers inside of the used programming language.

The ideal solution would be to get not a representation of a particular program but a representation of the calculus process that this program implies. Besides, it would be even more interesting to manipulate objects that would be totally independent of the language in which the program was given. It would only need to build a source-to-graph translator for each language.

In our system, a graph is built up from a program, which is a representation of the calculus process implied by the program. In the graph, the nodes represent the various operations of the calculus process (assignments, tests, inputs, outputs) and the arcs represent the flow-graph defined on these operations.

Example. Let us consider several sequences in different languages:

```
in FORTRAN:   DO 1 I = 1, N
            1 IF (MAX . LT . A(I)) MAX = A(I)
              ...

        or:   I = 1
            6 IF (MAX - A(I)) 4, 5, 5
            5 I = I + 1
              IF (I - N) 6, 6, 7
            4 MAX = A(I)
              GOTO 5
            7 ...
```

in EXEL:

$$I \leftarrow 1; \{\text{MAX} < A(I)\,?\,\text{MAX} \leftarrow A(I)|\ \text{¿}\ ; I \leftarrow I+1; I \leqslant N\ ?|!\ \text{¿}\}; \ldots$$

in ALGOL: $I := 1;$
TEST: *IF* $\text{MAX} < A(I)$ *THEN* $\text{MAX} := A(I);$
$I := I + 1;$
IF $I \leqslant N$ *THEN GOTO* TEST;
...

All of these sequences are translated into the same graph (see Fig. 1).

The use of graphs has another advantage: it is easy to find the predecessors of one node. Then, following the graph bottom-up is as easy as following it top-down. This property allows non-linear strategies that would not be possible if using linear representations.

Moreover, it is very convenient in a graph to apply program transformations. In particular, many syntactical transformations which are distinct in a linear algebraic language correspond to one and the same transformation in the graph.

The difference of structure between the graphs used by LAURA and the linear languages, as for instance the language used by Ruth, implies a fundamental difference of strategy:

—The system of Ruth uses a top-down analysis, matching two by two the actions of the PGM and of the student program (both are lists of actions). Then if it finds an important ('non-recoverable') difference, it does not deal any further with the rest of the student program. Firstly, this difference would perhaps be reducible using program transformations, but these are difficult to formalize and to activate in a linear language, especially since they are too numerous. So very few are used. Secondly, no information is obtained about the rest of the program.

—On the contrary the LAURA system may compare a region of one graph with any region of the other. Then it may identify several pairs of regions that

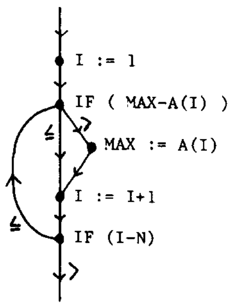


FIG. 1. The graph obtained from the sequences of Section 1.2.

calculate the same functions. Considering the regions that are unidentified, it may have information enough to apply powerful transformations advisedly, in order to make new identifications. At the end, the system is in a position to try to make several diagnostics of errors if several regions remain unidentified.

1.3. Program transformations

The comparison of programs may only be a possible way of debugging if the system can apply transformations when the structures of the two considered programs are different. In Section 1.2 we have explained why program transformations are easy to handle on a graph which represents a calculus process.

The transformations that may be used in LAURA are divided into two groups:

—The first ones are systematically applied to each graph in an attitude of standardization. These transformations may change the number of variables (some are added, some are removed). They may change the arithmetic expressions associated with nodes. They may change the structure of the graph. They may also correspond to a local analysis of the task that a subgraph makes, such as the resolution of induction equations (see Section 2.1). The object of these transformations is to increase the class of programs that a graph may represent.

—The second ones are used as the two graphs are matched. They are only applied in a blocked situation, if it makes new identifications possible. It would be harmful to apply these transformations systematically, either because they can be indefinitely applied (node splitting) or because the opposite transformation can also be applied (permutation, crossing over tests). So they are only activated by LAURA if heuristic conditions described in the second section are satisfied.

2. Description of the LAURA system

2.0. One current example

All along the description of the LAURA system we are going to follow one particular example. In order to make the work of the system easy to understand, it is a simple example, which does not represent a boundary for the system possibilities.

The exercise proposed is:

"A perfect number is a positive integer k which is equal to the sum of its divisors, 1 included and not k. Print out each perfect number less than or equal to 1000."

The programs given to the system are:

Program model

```
   DO 30 M=6,1000
   IS=1
   I=2
20 IF(M.NE.M/I*I)GOTO 21
   IS=IS+I+M/I
21 I=I+1
   IF(M-I*I)22,23,20
23 IS=IS+I
22 IF(M-IS)30,40,30
40 PRINT 41,M
30 CONTINUE
41 FORMAT(I5)
   STOP
   END
```

Student Program

```
   DO 60 N=3,1000
   I=2
   L=1
 1 IF(N-(N/I)*I)10,20,10
20 L=L+I+N/I
10 IF((I+1)**2-N)30,30,40
30 I=I+1
   GO TO 1
40 IF(L-N)50,60,50
50 WRITE(06,55)N
60 CONTINUE
   STOP
55 FORMAT(I4)
   END
```

2.1. Standardization

In order to simplify the further matching it is natural to normalize each program. It is well-known that there is no canonical form for programs. Yet we can reduce the search space by increasing the resemblance between the two objects to be compared.

The transformation itself of a program into a graph is a first and very important standardization. We have seen in Section 1.2 that the graph is a representation of the calculus process that a program implies.

Moreover, LAURA applies transformations to each graph in order to extend the standardization. Firstly, the arithmetic expressions are simplified using classic rewriting rules. Secondly, some transformations are systematically applied to each graph, which may change their structure.

Before describing the main ones, let us introduce three concepts:

—a variable is *defined* when it appears in an input list, or in the left member of an assignment.

—a variable is *used* when it appears in an output list, in the right member of an assignment, in a test, or as an array index.

—a *fuseau* is a quasi-strongly connected graph which has only one entry node and only one exit node [3].

2.1.1. *Variable separation*

When a variable X is defined at several places in the program, it is sometimes possible to determine that no use of one particular definition can be a use of another one. In this case, the system generates a new name for this definition and for all its uses.

Example.

$$\begin{array}{lcl} \text{READ } X & & \text{READ } X \\ Y = f(X) & \Rightarrow & Y = f(X) \\ X = A + B & & WX = A + B \\ Z = g(X) & & Z = g(WX) \end{array}$$

It is more difficult to establish such a condition when the variable is an array. In LAURA, arrays are never separated.

2.1.2. *Composition*

In order to remove intermediate variables (whose presence in only one program makes the matching more difficult) and to make calculus functions show up in each program, LAURA uses the following transformation called 'composition':
—Let D be a definition of X: $X = \text{Exp}$, such as there is no occurrence of X in Exp.
—Let U be a node in which X is used.

$\Rightarrow$ If it is impossible that the value of X used in U comes from a definition of X other than D, and if no variables used in Exp are redefined between D and U, the system changes X into Exp for every occurrence of X in U.

In Fig. 2, compositions are applied in two different subgraphs. They make the well-known formulas of second-order equations show up and increase the similarity of the two subgraphs. After normalization of the arithmetic expressions, the last difference will be removed by making $U = -B/2A$ cross over the test in the second subgraph and by suppressing it in the branches in which U is no longer used.

Let us note that, when a variable is an array, we must be careful of the values of the indexes before applying compositions.

Example.

$$A(1) = f_1(\cdots) \tag{1}$$

$$X = g(A(1)) \tag{2}$$

$$A(J) = f_2(\cdots) \tag{3}$$

$$Y = h(A(J)) \tag{4}$$

$$Z = A(1) + A(J) \tag{5}$$

It is possible to apply a composition in (2) which becomes:

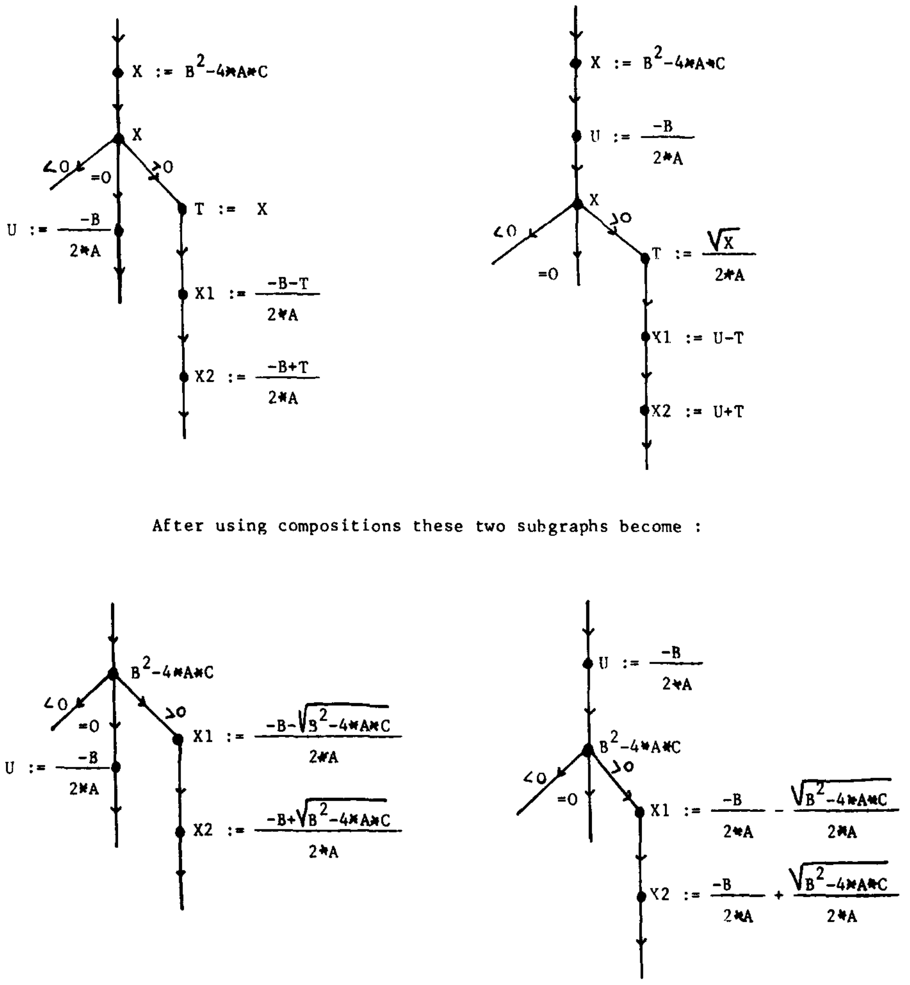
$$X = g(f_1(\cdots))$$

and in (4) which becomes:

$$Y = h(f_2(\cdots))$$

But, in (5), it is only possible to compose $A(J)$ because the value of $A(1)$ is $f_1(\cdots)$ if $J \neq 1$ and $f_2(\cdots)$ if $J = 1$. Thus (5) becomes:

$$Z = A(1) + f_2(\cdots).$$



After using compositions these two subgraphs become :

Note : useless definitions are removed by the system.

FIG. 2. Example of compositions.

2.1.3. *Independent calculus separation*

When there are several independent calculus processes in the same loop, it is possible to separate them if the following conditions are verified:
—It is possible to separate the body of the loop in two fuseaux F_1 and F_2.
—No variable defined in one fuseau is used in the other.
—No variable used in the exit test is defined in F_1 or F_2.
Then a transformation is applied that generates two loops (see Fig. 3).

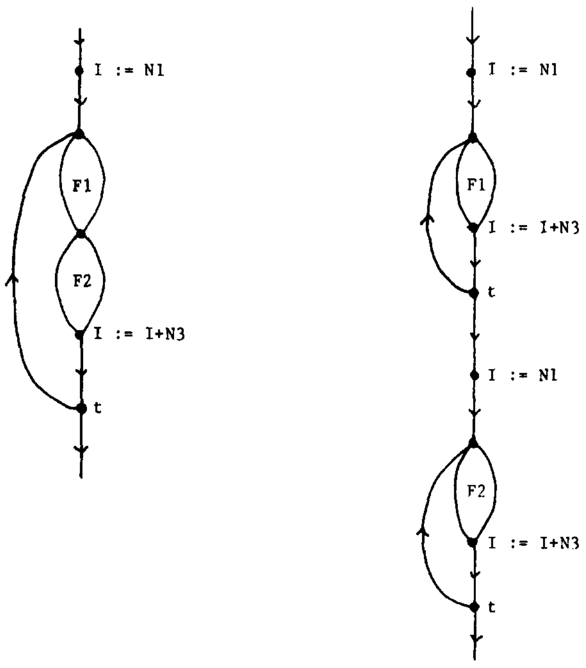
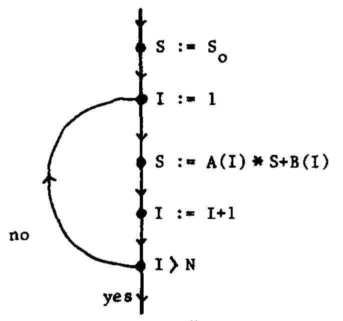


FIG. 3. Loop separation.

2.1.4. *Induction equations*

The linear induction equations of the first order may be solved to obtain a calculus function as shown in Fig. 4.

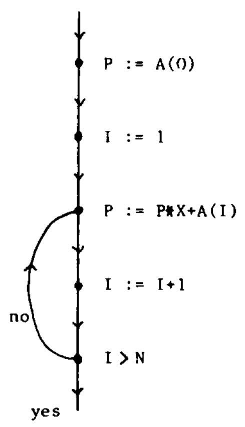
In the example of Fig. 5 two programs use two different methods for a



$$\Downarrow$$

$$S := \prod_{I=1}^{N} A(I) * S_0 + \sum_{I=1}^{N-1} (B(I) * \prod_{K=I+1}^{N} A(K)) + B(N)$$

FIG. 4. Resolution of a linear induction equation of the first order.



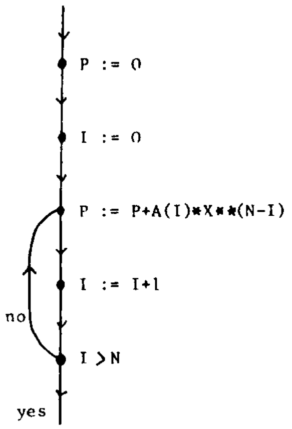
1 - The Method of HORNER

$$\Downarrow$$

$$P := \prod_{i=1}^{N} X*A(0) + \sum_{I=1}^{N-1} (A(I)* \prod_{K=I+1}^{N} X)+A(N)$$

$$\Downarrow$$

$$P := \sum_{I=0}^{N} A(I)*X**(N-1)$$



2 - The sum of the monomials

$$\Downarrow$$

$$P := 0 + \sum_{I=0}^{N} A(I)*X**(N-I)$$

FIG. 5. Two calculations of the same polynomial.

polynomial calculation. The resolution of the induction equations gives the same formula in the two cases. This example shows the power of this transformation which may extract the same formula from two sequences of program that have not used the same algorithm.

2.1.5. *Comparison with Program Optimization*

The transformations that LAURA systematically applies on a program are often the opposite of those to be applied in order to optimize it: our standardization eliminates most of the gains in memory space or in running time that the programmer had designed. The separation of variables introduces new memory space, the composition often implies repetitions of calculations and the separation of independent calculus increases the number of loops. Also, the solving of induction equations may completely destroy a subtle, efficient algorithm (such as Hörner's method in a polynomial calculation).

This opposition between the classic optimizations and our standardization is quite natural. Our purpose is the further matching of two graphs and it is interesting to get rid of particular subtleties.

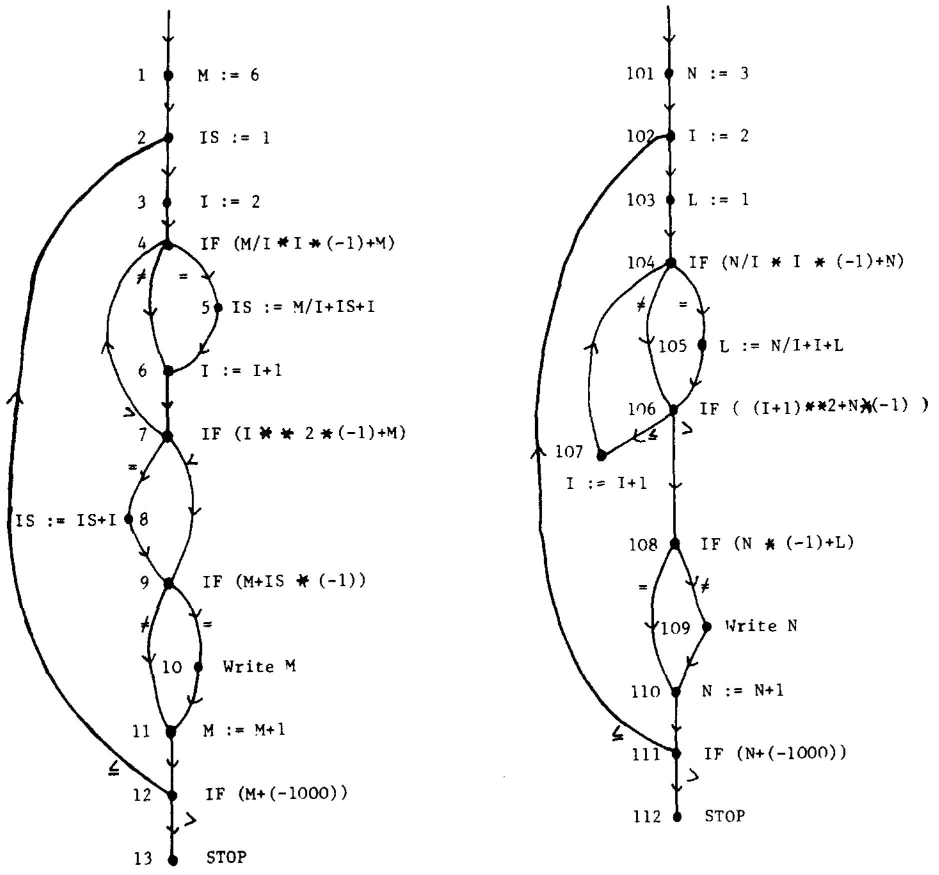


FIG. 6. The current example: The two graphs after standardizations.

2.1.6. *The current example*

From the two programs given in Section 2.0, the system has built two graphs. Fig. 6 shows these two graphs after standardization. Let us note the traduction of the DO instructions and of the boolean IF instructions, and the modifications in some arithmetic expressions (for example, in node 7, $I * I$ has been changed into $I ** 2$ and has become the first term of the sum $(I ** 2) * (-1) + M$ because it is more complex than the other term).

2.2. First step—first level of matching

In the beginning the system tries to show that the two graphs represent two calculus processes of the same functions. It tries to bind the variables, the nodes and the arcs. For this it uses one list of hypothesis and one list of pairs of nodes, called the 'working list'.

2.2.1. *The hypothesis list*

A hypothesis is made if there is a good chance of identifying a certain node s of the model graph with a certain node t of the student graph. Each hypothesis is numerically evaluated using a classic evaluation function that looks for the different characteristics of the two nodes: same connectives, same constants, already identified variables, equal number of free variables, etc. In this way, a number $P(s, t)$ is associated with each hypothesis $H(s, t)$ and is called its 'plausibility'. In the beginning, for each node of the model graph, plausibilities are evaluated for each node of the student graph which has the same nature. The four best ones are stored and the four corresponding hypothesis are put in the list. Each time the system identifies two variables, all the plausibilities of the hypothesis of the list are reevaluated. When it identifies two nodes the list of hypothesis is also updated. Furthermore, the fact that s and t are found to be immediate successors (or predecessors) of two identified nodes generates a new hypothesis $H(s, t)$ or, if it already exists, increases its plausibility.

The hypothesis and their plausibilities are used by the system to match the good pairs of nodes as soon as possible.

2.2.2. *Matching of two nodes*

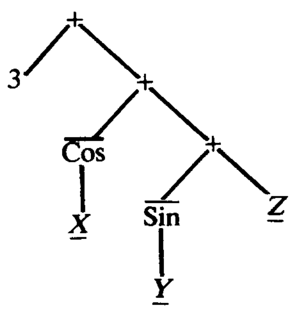
The LAURA system often has to match one node s of the model graph and one node t of the student graph. These nodes must have the same nature (for example two assignments) since the system matches them hoping to identify them. If it is not possible, the matching may still give useful information.

When matching two nodes, the most difficult parts to compare are the arithmetic expressions. To make it easier, all the arithmetic expressions are normalized when the system builds the graphs: rewriting rules are applied, sums and products are reduced and one order is defined between their terms.

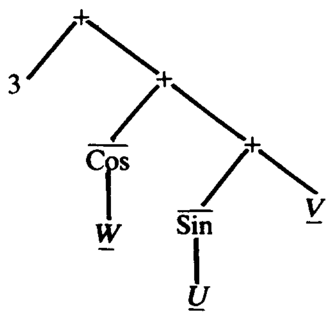
Example.

$$\text{Cos}\, X + 2\sin Y + Z + 3 - \text{Sin}\, Y$$

and Sin $U + V + 1 + \text{Cos}\, W + 2$ give the following lists:



and



and it is obvious that the comparison will be easier.

In order to match two arithmetic expressions, LAURA follows the two lists in parallel. If it finds the same connective, the same constant or two variables already identified with each other, it goes on. If a difference is met, it tries to apply a transformation rule (for instance a rule of distributiveness). If it fails, it generates one condition and starts again in matching the rest of the lists.

Finally the system returns either success or the list of the conditions that should be accepted to identify the nodes. It may also return failure if it has found a too complex condition.

The degree of complexity that the system accepts for a condition depends on the step. In the first step, when it tries to establish the equivalence of the two graphs, it only accepts one (and only one) condition $X \leftrightarrow Y$, where X and Y are free variables of G_M and G_E respectively. We shall see later, in Section 2.4 that more complex conditions may be considered in the second step.

Let us give now some examples of conditional successes at the first step:

Examples.

$$V = 1 - \text{Sin}\, X \quad \text{and} \quad V' = 1 - \text{Sin}\, X':$$

success at the first step (X and X', V and V' are already identified with each other).

$$V = 1 - \text{Sin}\, Y \quad \text{and} \quad V' = 1 - \text{Sin}\, Z:$$

conditional success at the first step if Y and Z are free variables. The condition is $Y \leftrightarrow Z$. Y and Z will be identified.

The matching of s and t may produce the identification of these two nodes. Also, it may lead in the first step to the identification of two variables X and Y, if the condition $X \leftrightarrow Y$ is accepted. In this case, all the plausibilities are recalculated to take in account this identification.

2.2.3. *The working list*

The system also uses a 'working list' that always contains the pairs (s, t) of identified nodes, for which at least one arc of extremity s (or t) is unidentified. Then the system can consider at each moment a pair (s, t) of the working list and try to identify two connected subgraphs around s and t respectively. For this, it matches two by two their immediate successors and predecessors. If a new pair of nodes is so identified, it matches the immediate successors and predecessors of these nodes, and so on.

When the exploration around a pair of the working list is finished, this pair either has vanished out of the list, or has received a mark. The system will start

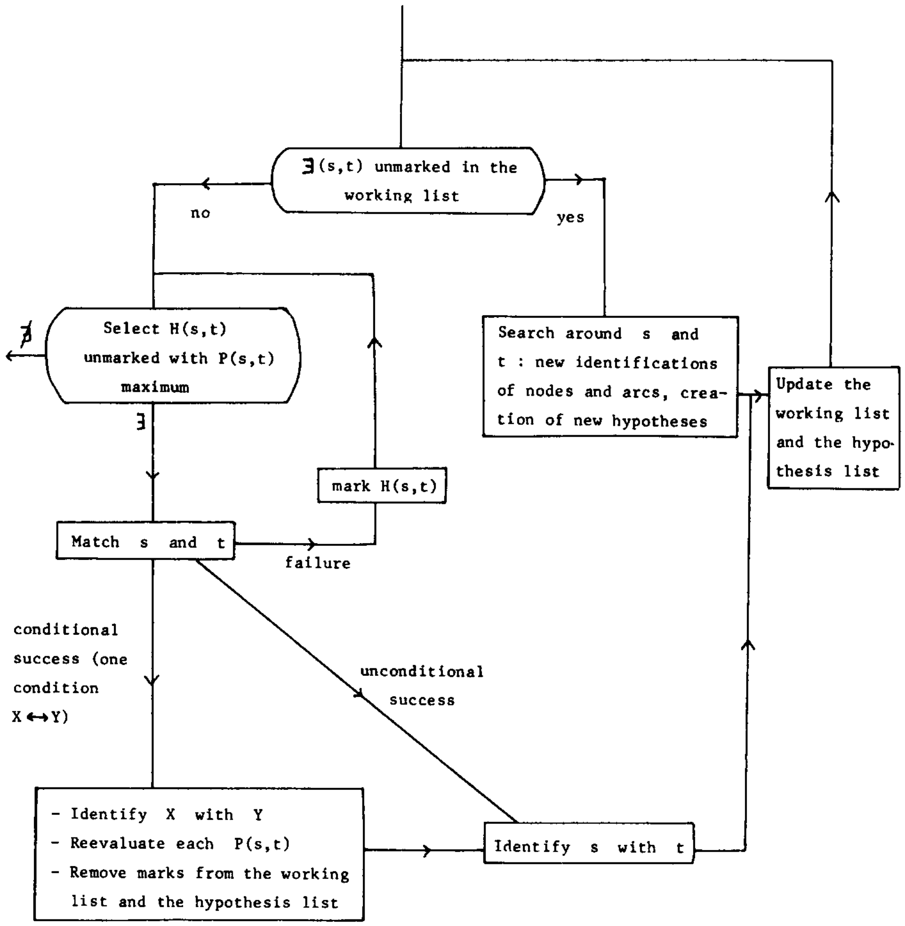

∃(s,t) unmarked in the working list
no
yes
Select H(s,t) unmarked with P(s,t) maximum
∄
∃
Search around s and t : new identifications of nodes and arcs, creation of new hypotheses
Update the working list and the hypothesis list
mark H(s,t)
Match s and t
failure
conditional success (one condition X↔Y)
unconditional success
- Identify X with Y
- Reevaluate each P(s,t)
- Remove marks from the working list and the hypothesis list
Identify s with t

FIG. 7. First step. First level of matching.

again on a new exploration around the following pair of the list. The use of graphs allows this non-linear strategy.

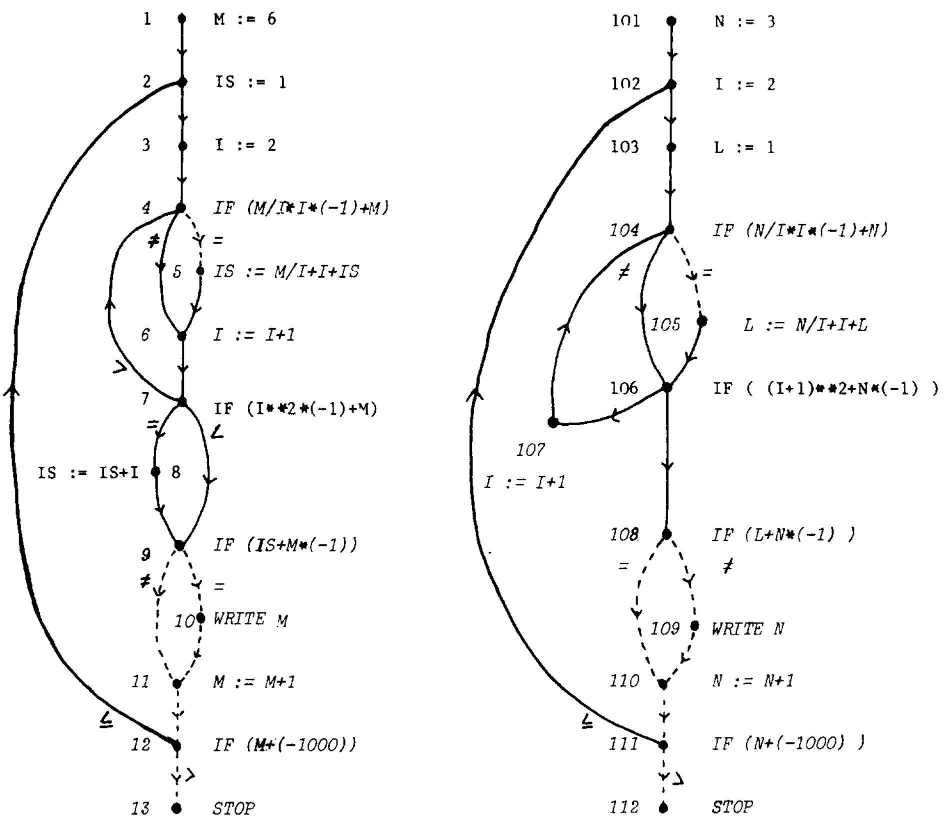
If the working list is empty or contains only marked pairs, the system considers the hypothesis which has the greatest plausibility and tries to identify two new points. If it succeeds, it again starts to identify new subgraphs.

All this work may be described by the diagram in Fig. 7.

2.3. First step—second level of matching

2.3.0. *The current example*

At the end of the first level, the state of the two graphs is given in Fig. 8.



- in italics : identified nodes
- dotted line : identified arcs

FIG. 8. The current example: The two graphs after the first level.

2.3.1. *Heuristic transformations of programs*

Even if the two programs calculate the same functions, the first level cannot succeed in identifying the two graphs. That would only be possible if the two graphs had the same structure, and this is in general not true.

In most cases it is necessary to apply transformations to the graphs in order to make their structures similar. These transformations, which cannot systematically be applied (as opposed to the standardization transformations), are subject to heuristic criteria. They are only applied if it makes the identification of new pairs of nodes possible. Then new elements appear in the working list, which the system will use for new exploration.

This second level is summarized by the diagram in Fig. 9.

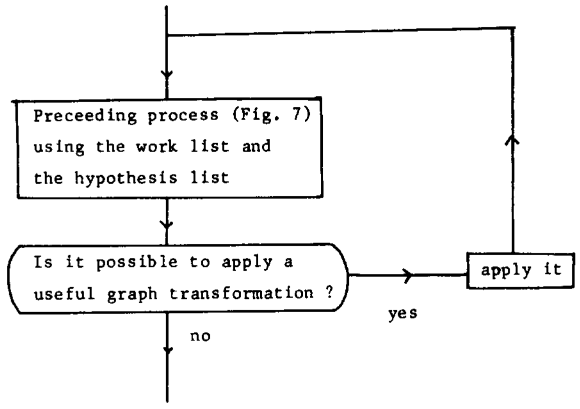
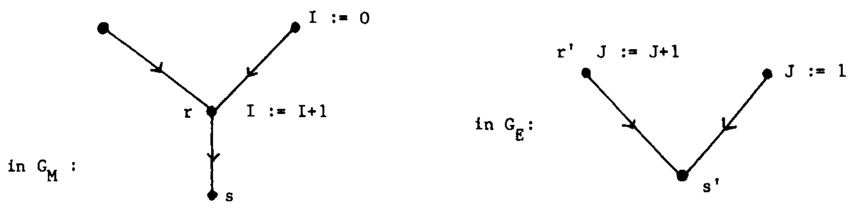

Preceeding process (Fig. 7) using the work list and the hypothesis list
Is it possible to apply a useful graph transformation ?
apply it
yes
no

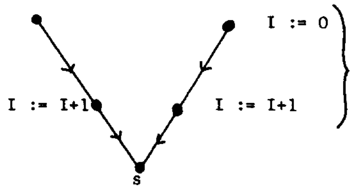
FIG. 9. Summarized diagram of one step.



Conditions :

- I and J are identified
- r and r' , s and s' are identified
- s has only one father, r
- r has two fathers
- s' has two fathers. One of them, r' is identified with r.

Transformation in G_M :



composed into I := 1, which will be identified with J := 1

Note : This transformation makes a program in which the increment of the index is made inside a loop similar to a program in which the increment is made when jumping back.

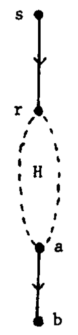
FIG. 10. Example of node splitting.

The main transformations that LAURA uses at this level are:
—node splitting,
—permutation (between two nodes or between a node and a subgraph),
—crossing over one test.

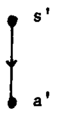
Some examples are given in Figs. 10, 11, and 12.

The transformations that LAURA uses at this level are very powerful. They are the essential tool in order to deal with programs in which the control structure is different from those of the program model.

in G_M :



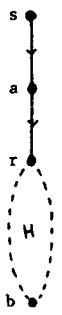
in G_E :



Conditions :

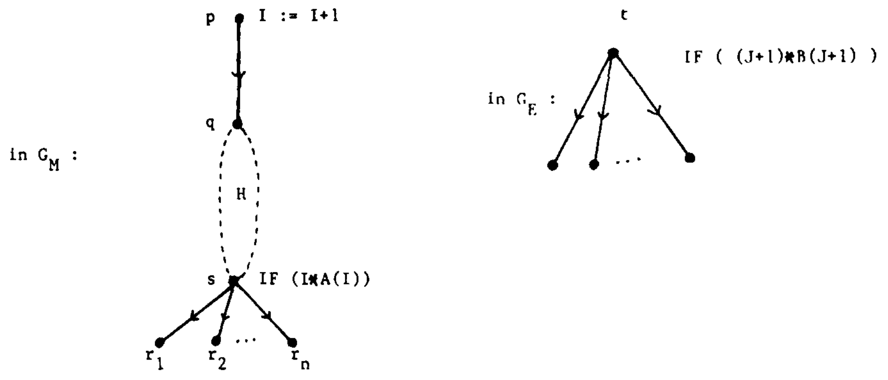
- s and s' , a and a' , are identified.
- a has only one son, b.
- the subgraph H has a structure of fuseau (see Section 2. 1. 3) and a is its output node.
- a is permutable with all the other nodes of H (we assume that two nodes are permutable if the variable defined in one is neither used, nor redefined in the other and if no variable is used in both nodes).

Transformation in G_M :



The arc (s,a) will be identified with the arc (s',a')

FIG. 11. Example of permutation: A successor is moved up to the place of a son.



Conditions :

- the variables I and J, A and B are identified

- there is an hypothesis H(s,t) and, in matching s and t, one condition has been obtained : I $\leftrightarrow$ J+1 := f(J)

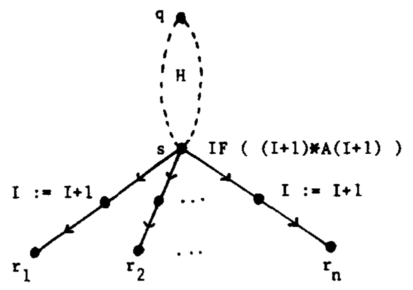
- p is an assignment I := f(I)

- p is a predecessor of s which is always executed before s

- H has a structure of fuseau

- all the nodes of H other than s are permutable with p (I neither used nor redefined)

Transformations in G_M :



s and t will be identified

FIG. 12. Example of crossing over one test that needs a composition.

2.4. Second step—diagnostics of errors

In the first step the goal of the LAURA system was to identify the two graphs entirely in order to conclude that the student program was correct. If this total identification was not possible, the system strives to reach another goal: to express diagnostics of errors.

The second step is in fact the same process as the first one, with a single but essential difference: In the first level of the step, conditional success in matching two points s and t may be obtained with more complex conditions.

As a matter of fact, the goal of the system at the second step is to detect errors. So, if it only finds one difference between two nodes such as a difference of variable, of constant, or operator,... the system assumes it has probably located one error. Then it identifies the two nodes despite the difference and as it put them in the working list, it will try to make new identifications.

Examples of conditional successes at the second step.

$$V = 1 - \mathrm{Sin}\, X \quad \text{and} \quad V' = 1 - \mathrm{Cos}\, X':$$

failure at the first step, conditional success at the second step (condition: Cos$\leftrightarrow$Sin).

$$V = 1000 \quad \text{and} \quad V' = 10000:$$

failure at the first step, conditional success at the second step (condition: 10000$\leftrightarrow$1000).

$$A(I, J) = A(I, K) + A(K, J) \quad \text{and} \quad A'(I', J') = A'(I', K') + A'(I', J'): \tag{1}$$

failure at the first step, conditional success at the second step (condition $I' \leftrightarrow K'$).

These new conditional successes, possible during the second step only, produce new identifications of nodes and, consequently, new explorations of the graphs. Of course when it makes a new identification, the system gives the user a *warning*: *s and t have been identified despite a specified difference.*

The difference may be of no importance (for example in (1), I' and K' may always have the same value). On the contrary it may reveal an error in the student implementation.

The main diagnostics the LAURA system can express are:

—Error of variable

ex: $S = A(I, J)$ and $S' = A'(I', K')$.

—Error of constant

ex: $N = 1000$ and $N' = 10000$.

—Unary connective forgotten

ex: $Y = 1 + \mathrm{Log}\,\mathrm{Sin}\, X$ and $Y' = 1 + \mathrm{Log}\, X'$.

—Unary connective useless

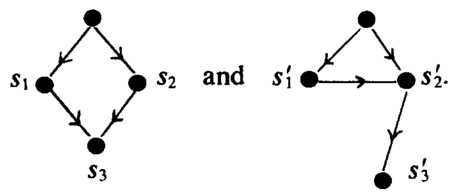
ex: $Y = 1 + \mathrm{Log}\, X$ and $Y' = 1 + \mathrm{Log}\,\mathrm{Sin}\, X'$.

—Error of sign

ex: $X = X + Y$ and $X' = X' - Y'$.

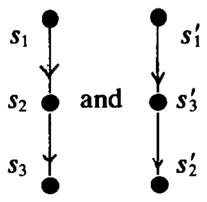
—Error of branching

ex:



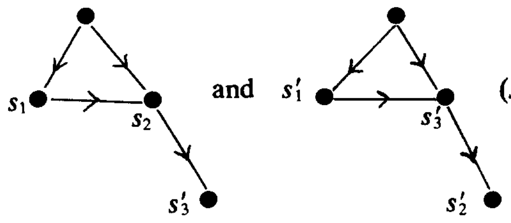
—Inversion of two instructions

ex:



(s'_3 and s'_2 not permutable).

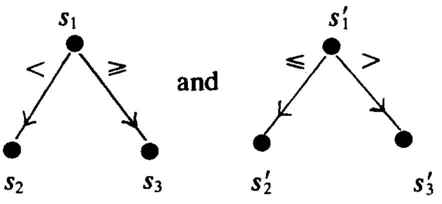
ex:



(s'_3 and s'_2 not permutable).

—Error of conditions on the arcs coming from one test

ex:



The conditions accepted in this second step determine whether or not th system is able to make certain diagnostics. The more complex the condition may be, the more sophisticated diagnostics may be possible. But of course it i impossible to accept just any condition: if the differences between two node are too large, it is unlikely that a local error is made at this node. Mor probably, the two matched nodes are not corresponding steps of the two calculus processes. The conditions accepted by LAURA in the second step must correspond to differences that have a good chance of revealing a local mistake.

Examples of conditions not accepted.

$$\left.\begin{array}{lll} V=1-\text{Sin}\,X & \text{and} & W'=1-\text{Sin}\,Y' \quad (W \text{ and } Y \text{ not free}) \\ V=1-\text{Sin}\,X & \text{and} & V'=X'-\text{Cos}\,X' \\ V=1-\text{Sin}\,X & \text{and} & V'=3X'+2! \end{array}\right\} \begin{array}{l}\text{failure}\\ \text{at both}\\ \text{steps.}\end{array}$$

When the second step is finished, it is possible that some subgraphs still remain unidentified. In this case *these subgraphs are printed out by the system.* In this way the user knows in which part of his implementation a doubt remains. If this part contains a semantic mistake, it has not been pointed out by the system but *it has been localized*, which is quite a big help in debugging.

The diagram of Fig. 13 shows the different steps of the LAURA system. Let us point out that when a difference is neglected, a new identification is made and the system starts again at the first step.

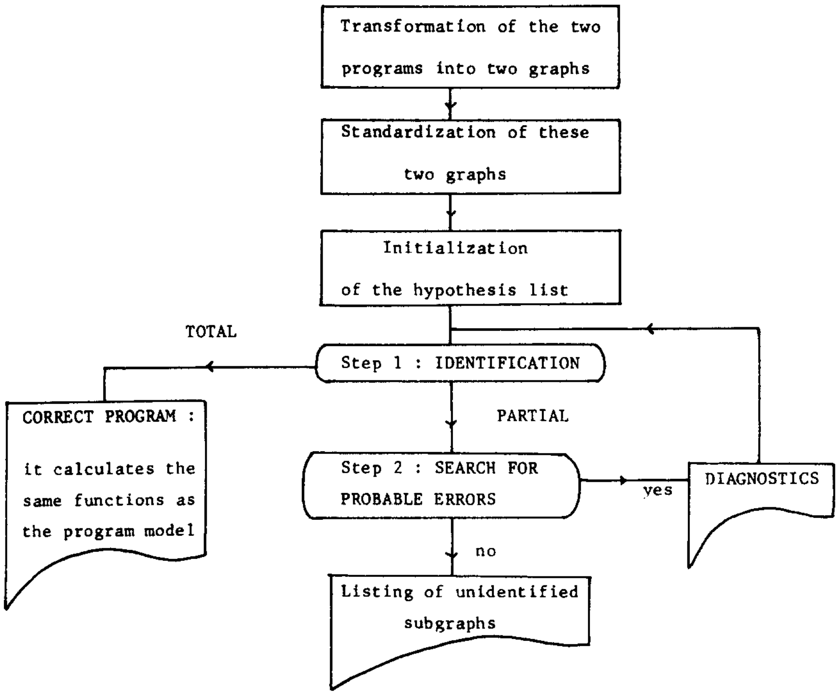

Transformation of the two programs into two graphs
Standardization of these two graphs
Initialization of the hypothesis list
TOTAL
Step 1 : IDENTIFICATION
CORRECT PROGRAM : it calculates the same functions as the program model
PARTIAL
Step 2 : SEARCH FOR PROBABLE ERRORS
yes
DIAGNOSTICS
no
Listing of unidentified subgraphs

FIG. 13. General flowchart of the system LAURA.

2.5. Results

2.5.0. *The current example*

After the first step, the two graphs were those of Fig. 14. During the second step, the system made the following actions:

(1) When matching the nodes 1 and 101, it has accepted the condition $3 \leftrightarrow 6$. It has identified the nodes 1 and 101 and the arcs (1, 3) and (101, 102). It has printed out a warning to inform the student. Let us note that the first perfect number is 6. The LAURA system has corrected an awkwardness in the student program.

(2) When looking at the conditions on the arcs coming from the nodes 9 and

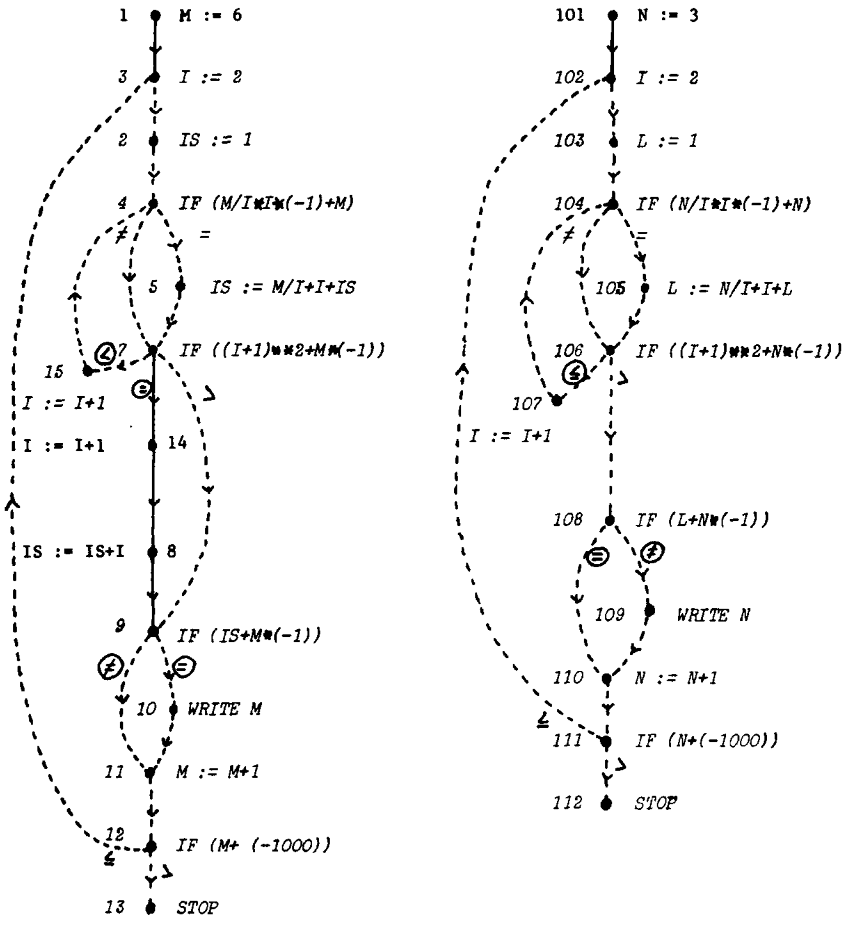


FIG. 14. The current example: The two graphs after the first step.

108, it has printed out a message to indicate that these conditions do not correspond. The LAURA system has detected a serious error: the student program prints out the not-perfect numbers.

(3) The nodes 8 and 14 in the model graph remain unidentified. The LAURA system has not been able to interpret this fact and has only printed a message. We discuss this point in the conclusion.

We give below the diagnostics printed out by the system:

```
DIAGNOSTICS
-----------


*** ATTENTION *** POUR IDENTIFIER LES INSTRUCTIONS  1 ET 101
*** ON A ADMIS L'EQUIVALENCE DE  6 ET 3
*** ERREUR POSSIBLE : VERIFIEZ LA VALEUR DE LA CTE OU DE LA VARIABLE QUE VOUS UTILISEZ

******************************

INSTRUCTION  8 DANS PROGRAMME 1 NON IDENTIFIEE

INSTRUCTION 14 DANS PROGRAMME 1 NON IDENTIFIEE

CONDITIONS DIFFERENTES SUR LES ARCS ISSUS DES INSTRUCTIONS  7 ET 106

CONDITIONS DIFFERENTES SUR LES ARCS ISSUS DES INSTRUCTIONS  9 ET 108

******************************



          COMPILE TIME=     3.10 SEC,EXECUTION TIME=     4.17 SEC,
```

(IBM 370–168)

2.5.1. *Experimentations*

The LAURA system has been tested with about a hundred programs written by students to solve eight different problems in various fields:
—management (taxes calculation, Electric company invoices),
—arithmetic (perfect numbers, Pascal's triangle),
—numerical analysis methods (integration, equation roots),
—sorting of an array A and N numbers (using a given algorithm: search for maximum and permutation).

LAURA has been written in FORTRAN and has about 7000 instructions. On an IBM 370/168 computer, the results have been obtained in a few seconds for the shortest problems (e.g. the perfect numbers) and in about thirty seconds for the longest one (Newton's method).

The program models and the student programs are all written in FORTRAN as we have built only one source-to-graph translator. The students are programming apprentices but their programs have been compiled successfully before.

Many programs not only have superficial differences (names of variables, arithmetic expressions,...) but also profound differences (structure, local algorithms,...). Most of the programs that calculate the same functions as the program model have been recognized as 'equivalent' by the system. In many others, LAURA has found errors and has expressed exact, clear diagnostics. In some other cases, LAURA has not identified the whole student program. It has only marked off one part that contains a difference with the model but it has not been able to interpret this difference. It was generally sufficient to make the student able to realize by himself either that he had made an error or that he had used a variation without consequences.

Thus the LAURA system may be a great help in debugging programs and in particular, an efficient tool for student programmers. It would be all the more useful since it could be used in an interactive, conversational way.

Given below are some examples of programs that LAURA has dealt with. For each example we present:
—the subject of the exercise proposed to the students,
—the program model and the student program, just as they are given to the system,
—the two programs generated from the model graph and from the student graph when the comparison is finished,
—the printed diagnostics.

Some comments are added to point out the main difficulties that were to be solved. When necessary, the consequences of detected errors or of non-interpreted differences are explained.

Example 1. Perfect numbers.

A perfect number is a positive integer *k* which is equal to the sum of its divisors, 1 included and not *k*. Print out each perfect number less than or equal to 1000.

Program model

```
C     0 NOMBRES PARFAITS - CORRIGE
      0 DO 100 I=6,1000
      0 IS=1
      0 K=2
      1 IF(I/K*K.EQ.I)IS=IS+K+I/K
      0 K=K+1
      0 IF(K*K.LT.I)GOTO 1
    100 IF(IS.EQ.I)WRITE(08,10)I
     10 FORMAT(I5)
      0 STOP
      0 END
```

Student program

```
C     0 NOMBRES PARFAITS - ETUDIANT 1
      0 N=6
      2 I=2
      0 L=1
      1 IF(N-(N/I)*I)10,20,10
     20 L=L+I+N/I
     10 IF((I+1)**2-N)30,40,40
     30 I=I+1
      0 GOTO 1
     40 IF(L-N)50,60,50
     60 WRITE(06,55)N
     55 FORMAT(I4)
     50 IF(N-1000)70,100,100
     70 N=N+1
      0 GOTO 2
    100 STOP
      0 END
```

```
*****************************
PROGRAMME DE REFERENCE
---------------------

     1 I=6
     3 K=2
     2 IS=1
     4 IF(I-I/K*K)07,05,07
     5 IS=I/K+IS+K
     7 IF(I-(K+1)**2)08,08,13
    13 K=K+1
       GOTO   4
     8 IF(IS-I)11,09,11
     9 PRINT I
    11 IF(1000-I)12,12,14
    14 I=I+1
       GOTO   3
    12 STOP
       END
```

```
*****************************
PROGRAMME ETUDIANT REECRIT PAR LE SYSTEME
-----------------------------------------

   101 N=6
   102 I=2
   103 L=1
   104 IF(N-N/I*I)106,105,106
   105 L=N/I+L+I
   106 IF(N-(I+1)**2)108,108,107
   108 IF(L-N)110,109,110
   109 PRINT N
   110 IF(1000-N)112,112,111
   112 STOP
   111 N=N+1
       GOTO 102
   107 I=I+1
       GOTO 104
       END
```

```
*****************************

DIAGNOSTICS
-----------

PROGRAMME CORRECT : IL CALCULE LES MEMES FONCTIONS QUE LE PROGRAMME DE REFERENCE

*****************************

COMPILE TIME=     3.17 SEC.EXECUTION TIME=     5.04 SEC.
```

(IBM 370-168)

In spite of many syntactical and structural differences, the LAURA system has determined that the student program is equivalent to the program model.

Example 2. Perfect numbers.

```
C      0 NOMBRES PARFAITS - CORRIGE
       0 DO 100 I=6,1000
       0 IS=1
       0 K=2
       1 IF(I/K*K.EQ.I)IS=IS+K+I/K
       0 K=K+1
       0 IF(K*K.LT.I)GOTO 1
     100 IF(IS.EQ.I)WRITE(08,10)I
      10 FORMAT(I5)
       0 STOP
       0 END
```

```
C-     0 NOMBRES PARFAITS - ETUDIANT 3
       0 N=3
      10 N=N+1
       0 I=2
       0 IS=1
      20 IF(N/I*I-N)21,22,21
      22 IS=IS+I+N/I
      21 I=I+1
       0 IF(I-N/2)20,23,23
      23 IF(N-IS)30,40,30
      40 WRITE(06,41)N
      41 FORMAT(I5)
      30 IF(N-1000)10,50,50
      50 STOP
       0 END
```

```
*****************************
PROGRAMME DE REFERENCE
---------------------

     1 I=6
     3 K=2
     2 IS=1
     4 IF(I/K*K-I)06,05,06
     5 IS=I/K+K+IS
     6 K=K+1
     7 IF(I-K**2)08,08,04
     8 IF(I-IS)11,09,11
     9 PRINT I
    11 IF(1000-I)12,12,13
    13 I=I+1
       GOTO   3
    12 STOP
       END
```

```
*****************************
PROGRAMME ETUDIANT REECRIT PAR LE SYSTEME
-----------------------------------------

   102 N=4
   103 I=2
   104 IS=1
   105 IF(N/I*I-N)107,106,107
   106 IS=N/I+I+IS
   107 I=I+1
   108 IF(N/2-I)109,109,105
   109 IF(N-IS)111,110,111
   110 PRINT N
   111 IF(1000-N)112,112,113
   112 STOP
   113 N=N+1
       GOTO 103
       END
```

```
*****************************

DIAGNOSTICS
-----------
*** ATTENTION *** POUR IDENTIFIER LES INSTRUCTIONS  1 ET 102
*** ON A ADMIS L'EQUIVALENCE DE  6 ET 4
*** ERREUR POSSIBLE : VERIFIEZ LA VALEUR DE LA CTE OU DE LA VARIABLE QUE VOUS UTILISEZ

*****************************

INSTRUCTION  7 DANS PROGRAMME 1 NON IDENTIFIEE

INSTRUCTION 108 DANS PROGRAMME 2 NON IDENTIFIEE

*****************************
COMPILE TIME=     3.12 SEC.EXECUTION TIME=     7.72 SEC.
```

These diagnostics show an awkward initialization of *N* and a mistake in the test of label 23: it is a serious error because some divisors will be added twice.

Example 3. Sorting.

In order to sort an array $A(1)\cdots A(N)$ in decreasing order, the following algorithm is used:

Step 1. Starting with $A(1)$, find the maximum and interchange it with $A(1)$.

Step 2. Do the same starting with $A(2)$, and so on until $N-1$.

```
C     0 TRI - CORRIGE
      0 DIMENSION A(100)
      0 READ 1,N,(A(I),I=1,N)
      1 FORMAT(I3/(8F10.2))
      0 NM=N-1
      0 DO 10 K=1,NM
      0 SUP=A(K)
      0 L=K
      0 KP=K+1
      0 DO 11 J=KP,N
      0 IF(SUP.GE.A(J))GOTO 11
      0 SUP=A(J)
      0 L=J
     11 CONTINUE
      0 A(L)=A(K)
     10 A(K)=SUP
      0 PRINT 2,(A(I),I=1,N)
      2 FORMAT(F10.2)
      0 STOP
      0 END
```

```
C     0 TRI - ETUDIANT 1
      0 DIMENSION A(10)
      0 READ 5,N,(A(I),I=1,N)
      0 NN=N-1
      0 DO 100 J=1,NN
      0 MAX=A(J)
      0 L=J
      0 DO 1 K=1,N
      0 IF(MAX.GE.A(K))GOTO 1
      0 MAX=A(K)
      0 L=K
      1 CONTINUE
      0 A(L)=A(J)
      0 A(J)=MAX
    100 CONTINUE
      0 PRINT 7,(A(I),I=1,N)
      7 FORMAT(F10.2)
      5 FORMAT(8F10.0)
      0 STOP
      0 END
```

```
*****************************
PROGRAMME DE REFERENCE
----------------------

     1 READ N
     2 READ A(KI),KI=1,N)
     3 NM=N-1
     4 K=1
     5 SUP=A(K)
     6 L=K
     8 J=K+1
     9 IF(A(J)-SUP)12,12,10
    12 J=J+1
    13 IF(J-N)09,09,14
    14 A(L)=A(K)
    15 A(K)=SUP
    16 K=K+1
    17 IF(K-NM)05,05,18
    18 PRINT A(LI),LI=1,N)
    19 STOP
    10 SUP=A(J)
    11 L=J
       GOTO  12
       END
```

```
*****************************
PROGRAMME ETUDIANT REECRIT PAR LE SYSTEME
-----------------------------------------

   101 READ N
   102 READ A(II),II=1,N)
   103 NN=N-1
   104 J=1
   105 MAX=A(J)
   106 L=J
   107 K=1
   108 IF(A(K)-MAX)111,111,109
   111 K=K+1
   112 IF(K-N)108,108,113
   113 A(L)=A(J)
   114 A(J)=MAX
   115 J=J+1
   116 IF(J-NN)105,105,117
   117 PRINT A(JI),JI=1,N)
   118 STOP
   109 MAX=A(K)
   110 L=K
       GOTO 111
       END
```

```
                   ****************************

                   VARIABLES IDENTIFIEES

                         N      N
                         A      A
                         SUP    MAX     *** ERREUR DE GENRE
                         J      K
                         KI     II
                         LI     JI
                         NM     NN
                         K      J
                         L      L

          ****************************

          DIAGNOSTICS
          -----------


          *** ATTENTION *** ERREUR(S) DE GENRE PROBABLE(S) :
              REPORTEZ-VOUS A LA LISTE DES IDENTIFICATIONS DE VARIABLES

          ****************************

          INSTRUCTION  8 DANS PROGRAMME 1 NON IDENTIFIEE

          INSTRUCTION 107 DANS PROGRAMME 2 NON IDENTIFIEE

          ****************************


          COMPILE TIME=     3.28 SEC.EXECUTION TIME=     4.70 SEC.
```

The nodes 8 and 107 remain unidentified. The inner loop in the student program must begin with $K = J + 1$ and not $K = 1$: it finds at each time the same maximum.

Example 4. Pascal's triangle.

Use the induction formula $C_n^p = C_{n-1}^p + C_{n-1}^{p-1}$ in order to compute the coefficients of Newton's binomial.

Starting with $A(1, 1) = A(2, 1) = A(2, 2) = 1$, compute and print out each line from 3 to 20.

(Given the weak representation capacity of our computer, use a real array A.)

```
C    0 TRIANGLE DE PASCAL - CORRIGE
     0 DIMENSION A(20,20)
     0 A(1,1)=1
     0 A(2,1)=1
     0 A(2,2)=1
     0 DO 1 I=3,20
     0 A(I,1)=1
     0 II=I-1
     0 DO 3 J=2,II
     3 A(I,J)=A(I-1,J)+A(I-1,J-1)
     0 A(I,I)=1
     1 PRINT 4,(A(I,J),J=1,I)
     4 FORMAT(20F6.0)
     0 STOP
     0 END
```

```
C    0 TRIANGLE DE PASCAL - ETUDIANT 1
     0 DIMENSION A(20,20)
     0 A(1,1)=1
     0 A(2,1)=1
     0 A(2,2)=1
     0 DO 1 I=3,20
     0 J=2
     0 A(I,1)=1
     3 A(I,J)=A(I-1,J)+A(I-1,J-1)
     0 J=J+1
     0 IF(J.EQ.I-1)GOTO 2
     0 GOTO 3
     2 A(I,I)=1
     1 PRINT 4,(A(I,J),J=1,I)
     4 FORMAT(20F6.0)
     0 STOP
     0 END
```

```
*****************************
PROGRAMME DE REFERENCE
----------------------

     1 A(1,1)=1
     2 A(2,1)=1
     3 A(2,2)=1
     4 I=3
     7 J=2
     5 A(I,1)=1
     8 A(I,J)=A(I-1,J-1)+A(I-1,J)
     9 J=J+1
    10 IF(J-I+1)08,08,11
    11 A(I,I)=1
    12 PRINT A(I,JI),JI=1,I)
    13 I=I+1
    14 IF(I-20)07,07,15
    15 STOP
       END
```

```
*****************************
PROGRAMME ETUDIANT REECRIT PAR LE SYSTEME
-----------------------------------------

   101 A(1,1)=1
   102 A(2,1)=1
   103 A(2,2)=1
   104 I=3
   105 J=2
   106 A(I,1)=1
   107 A(I,J)=A(I-1,J-1)+A(I-1,J)
   108 J=J+1
   109 IF(J-I+1)107,110,107
   110 A(I,I)=1
   111 PRINT A(I,II),II=1,I)
   112 I=I+1
   113 IF(I-20)105,105,114
   114 STOP
       END
```

```
*****************************
DIAGNOSTICS
-----------

*****************************

CONDITIONS DIFFERENTES SUR LES ARCS ISSUS DES INSTRUCTIONS 10 ET 109

*****************************

COMPILE TIME=     3.03 SEC,EXECUTION TIME=     3.44 SEC.
```

Serious error: the inner loop must also be executed when $J = I - 1$.

Example 5. Pascal's triangle.

```
C      0 TRIANGLE DE PASCAL - CORRIGE
       0 DIMENSION A(20,20)
       0 A(1,1)=1
       0 A(2,1)=1
       0 A(2,2)=1
       0 DO 1 I=3,20
       0 A(I,1)=1
       0 II=I-1
       0 DO 3 J=2,II
       3 A(I,J)=A(I-1,J)+A(I-1,J-1)
       0 A(I,I)=1
       1 PRINT 4,(A(I,J),J=1,I)
       4 FORMAT(20F6.0)
       0 STOP
       0 END
```

```
C      0 TRIANGLE DE PASCAL - ETUDIANT 4
       0 DIMENSION A(20,20)
       0 A(1,1)=1
       0 A(2,1)=1
       0 A(2,2)=1
       0 I=3
      14 J=2
       0 A(I,1)=1
      11 A(I,J)=A(I-1,J-1)+A(I-1,J)
       0 J=J+1
       0 IF(J.LT.I)GOTO 11
       0 A(I,J)=1
       0 PRINT 15,(A(I,J),J=1,I)
       0 I=I+1
       0 IF(I.LE.20)GOTO 14
       0 STOP
      15 FORMAT(1X,20F6.0)
       0 END
```

```
*****************************

PROGRAMME DE REFERENCE
----------------------

     1 A(1,1)=1
     2 A(2,1)=1
     3 A(2,2)=1
     4 I=3
     7 J=2
     5 A(I,1)=1
     8 A(I,J)=A(I-1,J-1)+A(I-1,J)
     9 J=J+1
    10 IF(I-J)11,11,08
    11 A(I,I)=1
    12 PRINT A(I,JI),JI=1,I)
    13 I=I+1
    14 IF(I-20)07,07,15
    15 STOP
       END
```

```
*****************************

PROGRAMME ETUDIANT REECRIT PAR LE SYSTEME
-----------------------------------------

   101 A(1,1)=1
   102 A(2,1)=1
   103 A(2,2)=1
   104 I=3
   105 J=2
   106 A(I,1)=1
   107 A(I,J)=A(I-1,J-1)+A(I-1,J)
   108 J=J+1
   109 IF(I-J)110,110,107
   110 A(I,J)=1
   111 PRINT A(I,II),II=1,I)
   112 I=I+1
   113 IF(I-20)105,105,114
   114 STOP
       END
```

```
****************************

DIAGNOSTICS
-----------


*** ATTENTION *** POUR IDENTIFIER LES INSTRUCTIONS 11 ET 110
*** ON A ADMIS L'EQUIVALENCE DE  I ET J
*** ERREUR POSSIBLE : VERIFIEZ LA VALEUR DE LA CTE OU DE LA VARIABLE QUE VOUS UTILISEZ

****************************


COMPILE TIME=     2.92 SEC.EXECUTION TIME=     4.04 SEC.
```

The system has printed out a warning. Yet, there is no error since J has always the same value as I when 110 is executed. We discuss this point in the conclusion.

Example 6. Trapezium method.

In order to get an approximation of the integral $\int_a^b f(x)\,dx$, we use the trapezium method. If one divides the interval $[a, b]$ into n intervals of equal length $h = (b-a)/n$, then:

$$S \simeq h \times \frac{f(a)+f(b)}{2} + \sum_{i=1}^{n-1} f(a+ih).$$

To ameliorate the precision, one may increase the number of trapeziums, for instance by doubling the value of n. Different approximated values of S will be so calculated until a relative precision ε is reached.

Values of a, b, n, ε are put in datas. The different values of S obtained must be printed out.

(In order to calculate $f(x)$, use a subrouting $FF(x)$.)

```
C     0 METHODE DES TRAPEZES - CORRIGE
      0 READ 100,A,B,N,EPSIL
    100 FORMAT(2F10.0,I10,F10.0)
      0 T=0.
      1 H=(B-A)/N
      0 S=0.
      0 NM=N-1
      0 DO 2 I=1,NM
      2 S=S+FF(A+I*H)
      0 S=H*((FF(A)+FF(B))/2+S)
      0 PRINT 200,S
    200 FORMAT(E15.6)
      0 IF(ABS((S-T)/S).LE.EPSIL)GOTO 1000
      0 T=S
      0 N=2*N
      0 GOTO 1
   1000 STOP
      0 END
```

```
C     0 METHODE DES TRAPEZES - ETUDIANT 1
      0 READ 1,A,B,N,C
      0 D=0
      4 H=(B-A)/N
      0 E=(FF(A)+FF(B))/2
      0 J=N-1
      0 DO 2 I=1,J
      2 E=E+FF(A+I*H)
      0 S=E*H
      0 PRINT 6,S
      0 IF(ABS((S-D)/S).LT.C)STOP
      0 D=S
      0 N=2*N
      0 GOTO 4
      1 FORMAT(2F10.0,I10,F10.0)
      6 FORMAT(E20.8)
      0 END
```

```
*****************************

PROGRAMME DE REFERENCE
----------------------


     1 READ A
     2 READ B
     3 READ N
     4 READ EPSIL
     5 T=0.
    13 W1S=(SIGMA(FF(A+(B-A)*MI/N),(MI=1.,N-1.))+FF(A)*0.5+FF(B)*0.5)*(B-A)/N
    14 PRINT W1S
    15 IF(EPSIL-ABS(((SIGMA(FF(A+(B-A)*MI/N),(MI=1.,N-1.))+FF(A)*0.5+FF(B)*0.5)*(B-A)/
       N-T)*N/(B-A)/(SIGMA(FF(A+(B-A)*MI/N),(MI=1.,N-1.))+FF(A)*0.5+FF(B)*0.5)))16,18,18
    18 STOP
    16 T=(SIGMA(FF(A+(B-A)*MI/N),(MI=1.,N-1.))+FF(A)*0.5+FF(B)*0.5)*(B-A)/N
    17 N=N*2
       GOTO  13
       END


*****************************

PROGRAMME ETUDIANT REECRIT PAR LE SYSTEME
-----------------------------------------


   101 READ A
   102 READ B
   103 READ N
   104 READ C
   105 D=0
   113 S=(SIGMA(FF(A+(B-A)*KI/N),(KI=1.,N-1.))+FF(A)*0.5+FF(B)*0.5)*(B-A)/N
   114 PRINT S
   115 IF(C-ABS(((SIGMA(FF(A+(B-A)*KI/N),(KI=1.,N-1.))+FF(A)*0.5+FF(B)*0.5)*(B-A)/N-D)
       *N/(B-A)/(SIGMA(FF(A+(B-A)*KI/N),(KI=1.,N-1.))+FF(A)*0.5+FF(B)*0.5)))117,117,116
   116 STOP
   117 D=(SIGMA(FF(A+(B-A)*KI/N),(KI=1.,N-1.))+FF(A)*0.5+FF(B)*0.5)*(B-A)/N
   118 N=N*2
       GOTO 113
       END


*****************************

DIAGNOSTICS
-----------


*****************************

CONDITIONS DIFFERENTES SUR LES ARCS ISSUS DES INSTRUCTIONS 15 ET 115

*****************************



COMPILE TIME=     3.64 SEC,EXECUTION TIME=    25.49 SEC.
```

Let us note that the two programs used different initializations. The resolution of the induction equations and, after, the compositions have translated the two calculus processes into the same formula.

Only one difference remains (*.LT.* instead of *.LE.* in the final test). It has no importance as trapezium method is an iterative method, but LAURA has not this knowledge.

Example 7. Trapezium method.

```
C     0 METHODE DES TRAPEZES - CORRIGE
      0 READ 100,A,B,N,EPSIL
    100 FORMAT(2F10.0,I10,F10.0)
      0 T=0.
      1 H=(B-A)/N
      0 S=0.
      0 NM=N-1
      0 DO 2 I=1,NM
      2 S=S+FF(A+I*H)
      0 S=H*((FF(A)+FF(B))/2+S)
      0 PRINT 200,S
    200 FORMAT(E15.6)
      0 IF(ABS((S-T)/S).LE.EPSIL)GOTO 1000
      0 T=S
      0 N=2*N
      0 GOTO 1
   1000 STOP
      0 END
```

```
C     0 METHODE DES TRAPEZES - ETUDIANT 3
      0 READ 1,A,B,K,EPS
      1 FORMAT(2F10.0,I10,F10.0)
      0 U=0
      6 PAS=(B-A)/K
      0 L=K-1
      0 SOMME=0
      0 I=1
      2 X=A+I*PAS
      0 F=FF(X)
      0 SOMME=SOMME+F
      0 I=I+1
      0 IF(I.LE.L)GOTO 2
      0 T=0.5*(FF(A)+FF(B))
      0 SOMME=SOMME+T
      0 SOMME=PAS*SOMME
      0 PRINT 3,SOMME
      3 FORMAT(E12.5)
      0 IF(ABS((SOMME-U)/U)-EPS)4,4,5
      4 STOP
      5 U=SOMME
      0 K=2*K
      0 GOTO 6
      0 END
```

```
*****************************

PROGRAMME DE REFERENCE
----------------------


     1 READ A
     2 READ B
     3 READ N
     4 READ EPSIL
     5 T=0.
     7 S=0.
     9 I=1
    10 S=FF(A+(B-A)*I/N)+S
    11 I=I+1
    12 IF(I-N+1)10,10,13
    13 W1S=(S+FF(B)*0.5+FF(A)*0.5)*(B-A)/N
    14 PRINT W1S
    15 IF(ABS(((S+FF(B)*0.5+FF(A)*0.5)*(B-A)/N-T)*N/(B-A)/(S+FF(B)*0.5+FF(A)*0.5))-
    18 STOP                                                    EPSIL)18,18,16
    16 T=(S+FF(B)*0.5+FF(A)*0.5)*(B-A)/N
    17 N=N*2
       GOTO   7
       END

*****************************

PROGRAMME ETUDIANT REECRIT PAR LE SYSTEME
-----------------------------------------


   101 READ A
   102 READ B
   103 READ K
   104 READ EPS
   105 U=0
   108 SOMME=0
   109 I=1
   112 SOMME=FF(A+(B-A)*I/K)+SOMME
   113 I=I+1
   114 IF(I-K+1)112,112,117
   117 W2SOMME=(SOMME+FF(B)*0.5+FF(A)*0.5)*(B-A)/K
   118 PRINT W2SOMME
   119 IF(ABS(((SOMME+FF(B)*0.5+FF(A)*0.5)*(B-A)/K-U)/U)-EPS)120,120,121
   121 U=(SOMME+FF(B)*0.5+FF(A)*0.5)*(B-A)/K
   122 K=K*2
       GOTO 108
   120 STOP
       END
```

```
DIAGNOSTICS
-----------
*****************************
INSTRUCTION 15 DANS PROGRAMME 1 NON IDENTIFIEE

INSTRUCTION 119 DANS PROGRAMME 2 NON IDENTIFIEE
*****************************

COMPILE TIME=    3.06 SEC.EXECUTION TIME=    12.87 SEC.
```

Serious error: the test

$$\left|\frac{\text{SOMME} - U}{U}\right| \leq \text{EPS}$$

is a division by 0 at the first time.

Example 8. Newton's method.

Let $P(x) = a_1x^n + a_2x^{n-1} + \cdots + a_{n+1}$ be a polynomial which has n real, distinct roots. Let x_0 be a value greater than the greatest root.

(1) Using Newton's method it is possible to calculate the greatest root (with a precision ε): a series that converges towards r is built from the formula

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

(2) Let r be the obtained value. Dividing $P(x)$ by $(x - r)$, a new polynomial is obtained with the same roots as P, r excepted. Again using Newton's method on the new polynomial, the second root of P is obtained.

Calculate successively the n real, distinct roots of P. Print out at each time all the values of the series x_n. To stop the calculation of this series, note that $f(x_n) \to 0$.

The degree N of the polynomial, x_0, ε and the coefficients of the polynomial are put in datas in this order.

```
C     0 NEWTON CORRIGE
      0 DIMENSION A(8)
      0 READ 2,N,T,R
      2 FORMAT(I10,2F10.0)
      0 M=N+1
      0 READ 3,(A(I),I=1,M)
      3 FORMAT(8F10.0)
     14 P=0
      0 M=N+1
      0 DO 6 I=1,M
      6 P=P*T+A(I)
      0 Q=0
      0 DO 4 I=1,N
      4 Q=Q*T+(N+1-I)*A(I)
      0 T=T-P/Q
      0 PRINT 8,T
      8 FORMAT(E17.7)
      0 IF(ABS(P)-R)12,12,14
     12 DO 16 I=2,N
     16 A(I)=A(I)+T*A(I-1)
      0 N=N-1
      0 IF(N-1)18,18,14
     18 X=-A(2)/A(1)
      0 PRINT 8,X
      0 STOP
      0 END
```

```
C     0  METHODE DE NEWTON    ETUDIANT 4
      0 DIMENSION A(8)
      0 READ 1,N,X,EPSIL
      1 FORMAT(I10,2F10.0)
      0 M=N+1
      0 READ 3,(A(I),I=1,M)
      3 FORMAT(8F10.0)
      2 Q=0
      0 DO 4 I=1,N,1
      4 Q=Q*X+(N+1-I)*A(I)
      0 P=0
      0 M=N+1
      0 DO 6 I=1,M,1
      6 P=P*X+A(I)
      0 X=X-(P/Q)
      0 PRINT 8,X
      8 FORMAT(E17.7)
      0 IF(ABS(P).GT.EPSIL)GOTO 2
      0 PRINT 8
      0 DO 7 I=2,N,1
      7 A(I)=A(I)+(X*A(I-1))
      0 IF(N.LE.2)GOTO 9
      0 N=N-1
      0 GOTO 2
      9 X=-A(2)/A(1)
      0 PRINT 8,X
      0 STOP
      0 END
```

```
*****************************

PROGRAMME DE REFERENCE
----------------------


     1 READ N
     2 READ T
     3 READ R
     4 W1M=N+1
     5 READ A(JJ),JJ=1,W1M)
     9 P=SIGMA(A(NJ)*T**(N-NJ+1.),(NJ=1.,N+1.))
    17 T=T-SIGMA(A(NJ)*T**(N-NJ+1.),(NJ=1.,N+1.))/SIGMA((N-JK+1.)*A(JK)*T**(N-JK),(JK=1.,N)
    18 PRINT T
    19 IF(ABS(P)-R)20,20,09
    20 I=2
    21 A(I)=A(I)+A(I-1)*T
    22 I=I+1
    23 IF(I-N)21,21,25
    25 IF(N-2)26,26,31
    31 N=N-1
       GOTO   9
    26 X=-A(2)/A(1)
    27 PRINT X
    28 STOP
       END


*****************************

PROGRAMME ETUDIANT REECRIT PAR LE SYSTEME
-----------------------------------------


   101 READ N
   102 READ X
   103 READ EPSIL
   104 W1M=N+1
   105 READ A(II),II=1,W1M)
   114 P=SIGMA(A(IJ)*X**(N-IJ+1.),(IJ=1.,N+1.))
   117 X=X-SIGMA(A(IJ)*X**(N-IJ+1.),(IJ=1.,N+1.))/SIGMA((N-LI+1.)*A(LI)*X**(N-LI),(LI=1.,N))
   118 PRINT X
   119 IF(ABS(P)-EPSIL)120,120,114
   120 I=2
   121 A(I)=A(I)+A(I-1)*X
   122 I=I+1
   123 IF(I-N)121,121,124
   124 IF(N-2)126,126,125
   126 W1X=-A(2)/A(1)
   127 PRINT W1X
   128 STOP
   125 N=N-1
       GOTO 114
       END


*****************************

DIAGNOSTICS
-----------

PROGRAMME CORRECT : IL CALCULE LES MEMES FONCTIONS QUE LE PROGRAMME DE REFERENCE

*****************************


         COMPILE TIME=     3.60 SEC.EXECUTION TIME=     37.83 SEC.
```

The student program is recognized equivalent, although the computations of $P(x)$ and $P'(x)$ are not made in the same order as in the program model.

Example 9. Newton's method.

```
C      0 NEWTON CORRIGE
       0 DIMENSION A(8)
       0 READ 2,N,T,R
       2 FORMAT(I10,2F10.0)
       0 M=N+1
       0 READ 3,(A(I),I=1,M)
       3 FORMAT(8F10.0)
      14 P=0
       0 M=N+1
       0 DO 6 I=1,M
       6 P=P*T+A(I)
       0 Q=0
       0 DO 4 I=1,N
       4 Q=Q*T+(N+1-I)*A(I)
       0 T=T-P/Q
       0 PRINT 8,T
       8 FORMAT(E17.7)
       0 IF(ABS(P)-R)12,12,14
      12 DO 16 I=2,N
      16 A(I)=A(I)+T*A(I-1)
       0 N=N-1
       0 IF(N-1)18,18,14
      18 X=-A(2)/A(1)
       0 PRINT 8,X
       0 STOP
       0 END
```

```
C      0  METHODE DE NEWTON   ETUDIANT 5
       0 DIMENSION A(20)
       0 READ 1,N,X,E
       0 M=N+1
       0 READ 15,(A(I),I=1,M)
       1 FORMAT(I10,2F10.0)
      15 FORMAT(8F10.0)
       7 F=A(1)
       0 I=1
       0 G=0
       4 F=F*X+A(I+1)
       0 G=G*X+(N-I+1)*A(I)
       2 I=I+1
       0 IF(I-N)4,4,3
       3 X=X-F/G
       0 PRINT 100,X
       0 IF(ABS(F)-E)5,5,7
       5 DO 10 I=2,N
      10 A(I)=A(I)+(X*A(I-1))
       9 N=N-1
       0 IF(N-1)12,12,7
      12 R=-A(2)/A(1)
       0 PRINT 100,R
     100 FORMAT(E17.7)
       0 STOP
       0 END
```

```
*****************************

PROGRAMME DE REFERENCE
----------------------

      1 READ N
      2 READ T
      3 READ R
      4 W1M=N+1
      5 READ A(KJ),KJ=1,W1M)
      9 P=SIGMA(A(MJ)*T**(N-MJ+1.),(MJ=1.,N+1.))
     17 T=T-SIGMA(A(MJ)*T**(N-MJ+1.),(MJ=1.,N+1.))/SIGMA((N-KK+1.)*A(KK)*T**(N-KK),(KK=1.,N))
     18 PRINT T
     19 IF(ABS(P)-R)20,20,09
     20 I=2
     21 A(I)=A(I)+A(I-1)*T
     22 I=I+1
     23 IF(I-N)21,21,24
     24 N=N-1
     25 IF(N-1)26,26,09
     26 X=-A(2)/A(1)
     27 PRINT X
     28 STOP
        END
*****************************

PROGRAMME ETUDIANT REECRIT PAR LE SYSTEME
-----------------------------------------

    101 READ N
    102 READ X
    103 READ E
    104 M=N+1
    105 READ A(II),II=1,M)
    109 F=SIGMA(A(JJ+1)*X**(N-JJ),(JJ=1.,N))+A(1)*X**N
    113 X=X-(SIGMA(A(JJ+1)*X**(N-JJ),(JJ=1.,N))+A(1)*X**N)/SIGMA((N-NI+1.)*A(NI)*X**(N-NI),
        (NI=1.,N))
    114 PRINT X
    115 IF(ABS(F)-E)116,116,109
    116 I=2
    117 A(I)=A(I)+A(I-1)*X
    118 I=I+1
    119 IF(I-N)117,117,120
    120 N=N-1
    121 IF(N-1)122,122,109
    122 R=-A(2)/A(1)
    123 PRINT R
    124 STOP
        END
```

```
DIAGNOSTICS
-----------


*****************************

INSTRUCTION   9 DANS PROGRAMME 1 NON IDENTIFIEE

INSTRUCTION  17 DANS PROGRAMME 1 NON IDENTIFIEE

INSTRUCTION 109 DANS PROGRAMME 2 NON IDENTIFIEE

INSTRUCTION 113 DANS PROGRAMME 2 NON IDENTIFIEE

*****************************

COMPILE TIME=     3.55 SEC,EXECUTION TIME=     37.78 SEC,
```

The student has used only one loop to compute $P(x)$ and $P'(x)$. This difference has been reduced by the system. Unfortunately the system could not identify the arithmetic expressions in (9, 109) and in (17, 113) since it is not able to prove that

$$u_1 + \sum_{i=1}^{n} u_{i+1} = \sum_{i=1}^{n+1} u_i.$$

Example 10. Newton's method.

```
C     0 NEWTON CORRIGE
      0 DIMENSION A(8)
      0 READ 2,N,T,R
      2 FORMAT(I10,2F10.0)
      0 M=N+1
      0 READ 3,(A(I),I=1,M)
      3 FORMAT(8F10.0)
     14 P=0
      0 M=N+1
      0 DO 6 I=1,M
      6 P=P*T+A(I)
      0 Q=0
      0 DO 4 I=1,N
      4 Q=Q*T+(N+1-I)*A(I)
      0 T=T-P/Q
      0 PRINT 8,T
      8 FORMAT(E17.7)
      0 IF(ABS(P)-R)12,12,14
     12 DO 16 I=2,N
     16 A(I)=A(I)+T*A(I-1)
      0 N=N-1
      0 IF(N-1)18,18,14
     18 X=-A(2)/A(1)
      0 PRINT 8,X
      0 STOP
      0 END
```

```
C     0 METHODE DE NEWTON   ETUDIANT 8
      0 DIMENSION A(10)
      0 READ 10,N,X,C
     10 FORMAT(I10,2F10.0)
      0 M=N+1
      0 READ 9,(A(I),I=1,M)
      9 FORMAT(8F10.0)
      1 P=0
      0 M=N+1
      0 DO 20 I=1,M
     20 P=P+A(I)*X**(N-I+1)
      0 Q=0
      0 DO 30 I=1,N
     30 Q=Q+A(I)*(N-I+1)*X**(N-I)
      3 X=X-P/Q
      0 PRINT 12,X
     12 FORMAT(F10.4)
      0 IF(ABS(P).GT.C)GOTO 1
      0 DO 5 I=2,N
      5 A(I)=A(I)+X*A(I-1)
      0 N=N-1
      0 IF(N.EQ.1)GOTO 6
      0 GOTO 1
      6 D=-A(2)/A(1)
      0 PRINT 12,D
      0 STOP
      0 END
```

```
PROGRAMME DE REFERENCE
----------------------


     1 READ N
     2 READ T
     3 READ R
     4 W1M=N+1
     5 READ A(JJ),JJ=1,W1M)
     9 P=SIGMA(A(NJ)*T**(N-NJ+1.),(NJ=1.,N+1.))
    17 T=T-SIGMA(A(NJ)*T**(N-NJ+1.),(NJ=1.,N+1.))/SIGMA((N-JK+1.)*A(JK)*T**(N-JK),(JK=1.,N))
    18 PRINT T
    19 IF(ABS(P)-R)20,20,09
    20 I=2
    21 A(I)=A(I)+A(I-1)*T
    22 I=I+1
    23 IF(I-N)21,21,24
    24 N=N-1
    25 IF(N-1)26,26,09
    26 X=-A(2)/A(1)
    27 PRINT X
    28 STOP
       END
******************************

PROGRAMME ETUDIANT REECRIT PAR LE SYSTEME
-----------------------------------------


   101 READ N
   102 READ X
   103 READ C
   104 W1M=N+1
   105 READ A(II),II=1,W1M)
   109 P=SIGMA(A(LI)*X**(N-LI+1.),(LI=1.,N+1.))
   117 X=X-SIGMA(A(LI)*X**(N-LI+1.),(LI=1.,N+1.))/SIGMA((N-IJ+1.)*A(IJ)*X**(N-IJ),(IJ=1.,N))
   118 PRINT X
   119 IF(ABS(P)-C)120,120,109
   120 I=2
   121 A(I)=A(I)+A(I-1)*X
   122 I=I+1
   123 IF(I-N)121,121,124
   124 N=N-1
   125 IF(N-1)109,126,109
   126 D=-A(2)/A(1)
   127 PRINT D
   128 STOP
       END
******************************

DIAGNOSTICS
-----------


******************************

CONDITIONS DIFFERENTES SUR LES ARCS ISSUS DES INSTRUCTIONS 25 ET 125

******************************


COMPILE TIME=     3.33 SEC.EXECUTION TIME=    33.84 SEC.
```

To compute $P(x)$ and $P'(x)$ the student has added all the monomials instead of using Horner's method. The system has reduced this difference. The only difference that remains is about tests 25 and 125. To prove that it is not really an error, the system should know that the input value of N is greater than 1.

Example 11. Second-order equation.

The parameters a, b, c, of the equation $ax^2 + bx + c = 0$ are put in data. Find the real roots. Print out their number and their values.

```
C     0 DISCUSSION EQUATION 2EME DEGRE DANS R : ETUDIANT 3
      0 READ(05,100)A,B,C
      0 IF(A.EQ.0)GOTO 1
      0 D=B**2-4*A*C
      0 IF(D)2,3,4
      4 NR=2
      0 D=SQRT(D)
      0 R1=(-B+D)/(2*A)
      0 R2=(-B-D)/(2*A)
      0 WRITE(66,200)NR,R1,R2
      0 STOP
      3 RU=-B/(2*A)
      7 NR=1
      0 WRITE(06,300)NR,R1
      0 STOP
      2 NR=0
      0 WRITE(06,400)NR
      1 IF(B)5,6,5
      6 R1=-B/A
      0 GO TO 7
      5 IF(C)2,8,2
      8 NR=-1
      0 GOTO 2
    100 FORMAT(3E12.4)
    200 FORMAT(I5)
    300 FORMAT(I5,E12.4)
    400 FORMAT(I5,2E12.4)
      0 STOP
      0 END
```

```
    124 NR=-1
DEFINITION NON UTILISEE... ELIMINEZ CETTE ANOMALIE ET SOUMETTEZ LE PROGRAMME DE NOUVEAU



COMPILE TIME=     3.08 SEC.EXECUTION TIME=     0.73 SEC.
```

Three times, the system has given a diagnostic of error a priori, when building the student graph.

First program: "The definition $NR = -1$ is never used".

Then, the student realizes that after $NR = -1$, the instruction WRITE (06,300) NR,$R1$ must be executed. So he changes the place of label 2.

```
C     0 DISCUSSION EQUATION 2EME DEGRE DANS R : ETUDIANT 3
      0 READ(05,100)A,B,C
      0 IF(A.EQ.0)GOTO 1
      0 D=B**2-4*A*C
      0 IF(D)2,3,4
      4 NR=2
      0 D=SQRT(D)
      0 R1=(-B+D)/(2*A)
      0 R2=(-B-D)/(2*A)
      0 WRITE(66,200)NR,R1,R2
      0 STOP
      3 RU=-B/(2*A)
      7 NR=1
      0 WRITE(06,300)NR,R1
      0 STOP
      0 NR=0
      2 WRITE(06,400)NR
      1 IF(B)5,6,5
      6 R1=-B/A
      0 GO TO 7
      5 IF(C)2,8,2
      8 NR=-1
      0 GOTO 2
    100 FORMAT(3E12.4)
    200 FORMAT(I5)
    300 FORMAT(I5,E12.4)
    400 FORMAT(I5,2E12.4)
      0 STOP
      0 END
      0
    119 NR=0
INSTRUCTION JAMAIS ATTEINTE... ELIMINEZ CETTE ANOMALIE ET SOUMETTEZ LE PROGRAMME DE NOUVEAU

COMPILE TIME=     2.98 SEC.EXECUTION TIME=     0.49 SEC.
```

Second program: "The instruction $NR = 0$ is never executed".

As a matter of fact. The student correction is false. He must not change label 2. He must create another label, 12.

```
   C     0 DISCUSSION EQUATION 2EME DEGRE DANS R : ETUDIANT 3
         0 READ(05,100)A,B,C
         0 IF(A.EQ.0)GOTO 1
         0 D=B**2-4*A*C
         0 IF(D)2,3,4
         4 NR=2
         0 D=SQRT(D)
         0 R1=(-B+D)/(2*A)
         0 R2=(-B-D)/(2*A)
         0 WRITE(66,200)NR,R1,R2
         0 STOP
         3 RU=-B/(2*A)
         7 NR=1
         0 WRITE(06,300)NR,R1
         0 STOP
         2 NR=0
        12 WRITE(06,400)NR
         1 IF(B)5,6,5
         6 R1=-B/A
         0 GO TO 7
         5 IF(C)2,8,2
         8 NR=-1
         0 GOTO 12
       100 FORMAT(3E12.4)
       200 FORMAT(I5)
       300 FORMAT(I5,E12.4)
       400 FORMAT(I5,2E12.4)
         0 STOP
         0 END
  115 RU=-B/(2*A)
DEFINITION NON UTILISEE... ELIMINEZ CETTE ANOMALIE ET SOUMETTEZ LE PROGRAMME DE NOUVEAU

    COMPILE TIME=     2.97 SEC.EXECUTION TIME=     0.79 SEC.
```

Third program: "The definition $RU = -B/(2A)$ is never used".

Here the system detects a punch-error, RU instead of $R1$.

Once this error has been corrected, the graph is constructed by LAURA without any other diagnostic and the matching with the program model is possible.

```
C     0 DISCUSSION EQUATION 2EME DEGRE DANS R : ''CORRIGE''
      0 READ(05,11)A,B,C
     11 FORMAT(3E12.4)
      0 IF(A)51,50,51
     51 D=B**2-4*A*C
      0 IF(D)1,2,3
      1 N=0
     70 WRITE(06,12)N
     12 FORMAT(I5)
      0 GO TO 100
      2 X0=-B/(2*A)
     20 N=1
      0 WRITE(06,13)N,X0
     13 FORMAT(I5,E12.4)
      0 GO TO 100
      3 X1=(-B-SQRT(D))/(2*A)
      0 X2=(-B+SQRT(D))/(2*A)
      0 N=2
      0 WRITE(06,14)N,X1,X2
     14 FORMAT(I5,2E12.4)
      0 GO TO 100
     50 IF(B)61,60,61
     61 X0=-C/B
      0 GO TO 20
     60 IF(C)1,62,1
     62 N=-1
      0 GO TO 70
    100 STOP
      0 END
```

```
C     0 DISCUSSION EQUATION 2EME DEGRE DANS R : ETUDIANT 3
      0 READ(05,100)A,B,C
      0 IF(A.EQ.0)GOTO 1
      0 D=B**2-4*A*C
      0 IF(D)2,3,4
      4 NR=2
      0 D=SQRT(D)
      0 R1=(-B+D)/(2*A)
      0 R2=(-B-D)/(2*A)
      0 WRITE(66,200)NR,R1,R2
      0 STOP
      3 R1=-B/(2*A)
      7 NR=1
      0 WRITE(06,300)NR,R1
      0 STOP
      2 NR=0
     12 WRITE(06,400)NR
      1 IF(B)5,6,5
      6 R1=-B/A
      0 GO TO 7
      5 IF(C)2,8,2
      8 NR=-1
      0 GOTO 12
    100 FORMAT(3E12.4)
    200 FORMAT(I5)
    300 FORMAT(I5,E12.4)
    400 FORMAT(I5,2E12.4)
      0 STOP
      0 END
```

```
PROGRAMME DE REFERENCE
----------------------

  1 READ A
  2 READ B
  3 READ C
  4 IF(A)06,19,06
 19 IF(B)20,21,20
 21 IF(C)07,22,07
 22 N=-1
  8 PRINT N
 23 STOP
  7 N=0
    GOTO   8
 20 X0=-C/B
 11 PRINT 1
 12 PRINT X0
    GOTO  23
  6 IF(B**2.+A*C*(-4.))07,09,14
 14 X2=((B**2.+A*C*(-4.))**0.5-B)/A*0.5
 13 X1=(-(B**2.+A*C*(-4.))**0.5-B)/A*0.5
 16 PRINT 2
 18 PRINT X2
 17 PRINT X1
    GOTO  23
  9 X0=B/A*(-0.5)
    GOTO  11
    END
```

```
PROGRAMME ETUDIANT REECRIT PAR LE SYSTEME
-----------------------------------------

101 READ A
102 READ B
103 READ C
104 IF(A)106,121,106
121 IF(B)123,122,123
122 R1=-B/A
117 PRINT 1
118 PRINT R1
114 STOP
123 IF(C)119,124,119
124 NR=-1
120 PRINT NR
    GOTO 121
119 NR=0
    GOTO 120
106 IF(B**2.+A*C*(-4.))119,115,109
109 W1R1=((B**2.+A*C*(-4.))**0.5-B)/A*0.5
110 R2=(-(B**2.+A*C*(-4.))**0.5-B)/A*0.5
111 PRINT 2
112 PRINT W1R1
113 PRINT R2
    GOTO 114
115 R1=B/A*(-0.5)
    GOTO 117
    END
```

```
DIAGNOSTICS
-----------


*****************************

INSTRUCTION 20 DANS PROGRAMME 1 NON IDENTIFIEE  }
                                                 }  (1)
INSTRUCTION 122 DANS PROGRAMME 2 NON IDENTIFIEE }

ERREUR DE BRANCHEMENT PROBABLE : ARC(120,121) AU LIEU DE ARC(120,114)      (2)

CONDITIONS DIFFERENTES SUR LES ARCS ISSUS DES INSTRUCTIONS 19 ET 121       (3)

*****************************


COMPILE TIME=     3.03 SEC,EXECUTION TIME=     7.55 SEC.
```

(1) $R1 = -B/A$ instead of $R1 = -C/B$.
(2) A stop has been forgotten.
(3) Labels 122 and 123 are interchanged.

Example 12. Electric Company Invoices.

For each consumer, the amount to be paid depends on the difference between the actual meter reading and the precedent one.

The consumption is divided in two sections by a certain limit.

—For the amount consumed up to the limit a first tariff is applied.

—For the rest a second cheaper tariff is applied.

Added to the sum thus obtained is the rent of the meter itself.

On a first card, are punched the price of this rent, the two tariffs and the limiting amount.

For each consumer there is one card with his consumer number, his precedent meter reading and the actual one. Print out the consumer number, the part of the consumption rated at the first tariff, the part rated at the second and the total amount to be paid.

N.B.: An end of file card contains 0 as consumer number.

```
C     0 FACTURE E.D.F. - CORRIGE
      0 READ 1,C,P1,P2,IBORNE
      1 FORMAT(3F10.2,I10)
     10 READ 11,NUM,NR,IAR
     11 FORMAT(3I10)
      0 IF(NUM.EQ.0)STOP
      0 ICONS=NR-IAR
      0 IF(ICONS.GT.IBORNE)GOTO 5
      0 M1=ICONS
      0 M2=0
      0 GOTO 6
      5 M1=IBORNE
      0 M2=ICONS-IBORNE
      6 TOT=C+M1*P1+M2*P2
      0 PRINT 20,NUM,M1,M2,TOT
     20 FORMAT(3I10,F10.2)
      0 GOTO 10
      0 END
```

```
C     0 FACTURE E.D.F. - ETUDIANT 2
      0 READ 1,CL,Q1,Q2,IT
      1 FORMAT(3F10.2,I10)
      0 READ 2,N,R2,R1
      2 FORMAT(I10,2F10.0)
      0 IF(N.EQ.0)GOTO 10
      0 IF(R2-R1-IT)3,3,4
      3 M2=0
      0 M1=R2-R1
      0 GOTO 5
      4 M2=R2-R1-IT
      0 M1=IT
      5 P1=M1*Q1
      0 P2=M2*Q2
      0 F=P1+P2+CL
      0 PRINT 4,N,M1,M2,F
      4 FORMAT(3I10,F10.2)
     10 STOP
      0 END
```

```
PROGRAMME DE REFERENCE                     PROGRAMME ETUDIANT REECRIT PAR LE SYSTEME
----------------------                     -----------------------------------------


     1 READ C                                  101 READ CL
     2 READ P1                                 102 READ Q1
     3 READ P2                                 103 READ Q2
     4 READ IBORNE                             104 READ IT
     5 READ NUM                                105 READ N
     6 READ NR                                 106 READ R2
     7 READ IAR                                107 READ R1
     8 IF(NUM)11,09,11                         108 IF(N)109,121,109
     9 STOP                                    121 STOP
    11 IF(NR-IAR-IBORNE)13,13,15               109 IF(R2-R1-IT)110,110,112
    15 M2=NR-IAR-IBORNE                        112 M2=R2-R1-IT
    14 M1=IBORNE                               113 M1=IT
    16 TOT=C+P2*M2+P1*M1                       116 F=CL+Q2*M2+Q1*M1
    17 PRINT NUM                               117 PRINT N
    18 PRINT M1                                118 PRINT M1
    19 PRINT M2                                119 PRINT M2
    20 PRINT TOT                               120 PRINT F
       GOTO   5                                    GOTO 121
    13 M2=0                                    110 M2=0
    12 M1=NR-IAR                               111 M1=R2-R1
       GOTO  16                                    GOTO 116
       END                                         END


*****************************

VARIABLES IDENTIFIEES

      C      CL
      P1     Q1
      P2     Q2
      IBORNE IT
      NUM    N
      NR     R2      *** ERREUR DE GENRE
      IAR    R1      *** ERREUR DE GENRE
      M1     M1
      M2     M2
      TOT    F


*****************************

DIAGNOSTICS
-----------


*** ATTENTION *** ERREUR(S) DE GENRE PROBABLE(S) :
    REPORTEZ-VOUS A LA LISTE DES IDENTIFICATIONS DE VARIABLES

*****************************

ERREUR DE BRANCHEMENT PROBABLE : ARC(120,121) AU LIEU DE ARC(120,105)

*****************************


COMPILE TIME=     3.47 SEC.EXECUTION TIME=     7.07 SEC.
```

The student only computes the amount for the first consumer.

Example 13. Electric Company Invoices.

```
C     0 FACTURE E.D.F. - CORRIGE
      0 READ 1,C,P1,P2,IBORNE
      1 FORMAT(3F10.2,I10)
     10 READ 11,NUM,NR,IAR
     11 FORMAT(3I10)
      0 IF(NUM.EQ.0)STOP
      0 ICONS=NR-IAR
      0 IF(ICONS.GT.IBORNE)GOTO 5
      0 M1=ICONS
      0 M2=0
      0 GOTO 6
      5 M1=IBORNE
      0 M2=ICONS-IBORNE
      6 TOT=C+M1*P1+M2*P2
      0 PRINT 20,NUM,M1,M2,TOT
     20 FORMAT(3I10,F10.2)
      0 GOTO 10
      0 END
```

```
C     0 FACTURE E.D.F. -ETUDIANT 5
      0 READ 1,X,Y,Z,K
      1 FORMAT(3F10.2,I10)
     50 READ 2,NC,I,J
      2 FORMAT(3I10)
      0 IF(NC.EQ.0)GOTO 100
      0 T=0
      0 L=I-J
      0 IF(L-K)20,20,30
     20 N1=L
      0 N2=0
      0 GOTO 40
     30 N1=K
      0 N2=L-K
     40 T=T+X+Y*N1+Z*N2
      0 PRINT 90,NC,N1,N2,T
     90 FORMAT(3I10,F10.2)
      0 GOTO 50
    100 STOP
      0 END
```

```
******************************

PROGRAMME DE REFERENCE
----------------------

      1 READ C
      2 READ P1
      3 READ P2
      4 READ IBORNE
      5 READ NUM
      6 READ NR
      7 READ IAR
      8 IF(NUM)11,09,11
      9 STOP
     11 IF(NR-IAR-IBORNE)12,12,14
     14 M1=IBORNE
     15 M2=NR-IAR-IBORNE
     16 TOT=C+P2*M2+P1*M1
     17 PRINT NUM
     18 PRINT M1
     19 PRINT M2
     20 PRINT TOT
        GOTO   5
     12 M1=NR-IAR
     13 M2=0
        GOTO  16
        END
```

```
******************************

PROGRAMME ETUDIANT REECRIT PAR LE SYSTEME
-----------------------------------------

    101 READ X
    102 READ Y
    103 READ Z
    104 READ K
    105 READ NC
    106 READ I
    107 READ J
    108 IF(NC)111,121,111
    121 STOP
    111 IF(I-J-K)112,112,114
    114 N1=K
    115 N2=I-J-K
    116 T=X+Z*N2+Y*N1
    117 PRINT NC
    118 PRINT N1
    119 PRINT N2
    120 PRINT T
        GOTO 105
    112 N1=I-J
    113 N2=0
        GOTO 116
        END
```

```
******************************

DIAGNOSTICS
-----------

PROGRAMME CORRECT : IL CALCULE LES MEMES FONCTIONS QUE LE PROGRAMME DE REFERENCE

******************************
```

The useless initialization of *T* is eliminated by compositions.

```
COMPILE TIME=      3.08 SEC.EXECUTION TIME=      6.12 SEC.
```

3. Further developments and conclusion

The main direction to improve the precision of the diagnostics given by our system is probably the addition of a theorem prover.

We wanted to study what debugging was possible without using assertions methods and, consequently, we have not used any theorem prover. Yet the very strategy of the LAURA system makes it able to generate in a natural way conjectures that it would be very interesting to prove. As a matter of fact, LAURA makes only diagnostics of possible errors. It makes one as it detects a certain difference and the difference itself may produce a question. Let us consider the Example 5 in Section 2.5.1. After the matching, there remains only one difference between the model graph and the student graph: $A(I, I) = 1$ in the model and $A(I, J) = 1$ in the student graph. Then it is easy to generate the proposition: "Is the value of J always equal to the value of I at this node?". The proof that this conjecture is true may be obtained without more information than the student graph.

In some cases such as in this example, this method would make the system able to neglect differences that correspond to a variation without consequences (in its actual version LAURA prints out a useless warning). In other cases the conjecture could be proven false. Then the system would print out a diagnostic of undoubted error. As useful conjectures may be generated easily, a theorem prover would be an efficient tool to make our system interpret differences before giving diagnostics.

It must be pointed out that without any more information than the program model and the student program, some differences cannot be interpreted. It would need, besides the knowledge of the task the program has to perform, a great knowledge of the field in which this task has a meaning. For example, we have seen in Section 2.5.0 about the current example, that after matching, nodes 8 and 14 remain unidentified in the model graph (Fig. 14).

This difference is very difficult to interpret. If an integer I divides the integer N, the quotient N/I is also a divisor of N. It is a well-known property in Arithmetic and the authors of the programs have both used it. They only look for the divisors less than or equal to $\sqrt{N}$ and for each they add I and N/I at the same time. In the program model, it was thought that $\sqrt{N}$ may be an integer. If it is, for the value $\sqrt{N}$ of I, only I must be added (otherwise $\sqrt{N}$ would be added twice). That is why there is a particular branch coming from the exit test 7. In the student program such a branch does not exist: the sum of the divisors is then wrong if N is a square number!! Can we conclude that the student has made a mistake? Not necessarily. In fact, it depends on whether or not this difference changes the printed out results, that means whether or not there are square numbers less than 1000 which are also perfect numbers!.... By chance there is the following theorem in Arithmetic: "no integer can be both a square number and a perfect number". So we can now conclude that the student program will print out the same result as the model....

This example shows that the automatic interpretation of differences sometimes requires a thorough knowledge of the field in which the problem is formulated. Thus, according to automatic debugging, we must choose between two possibilities. The first one is to debug programs in various fields, leaving the user himself to make some difficult interpretations. In our system, this solution has been selected. The second one is to give a good knowledge of a specialized field and to debug programs in this field only.

In the latest perspective we meet one of the most important general problems in Artificial Intelligence at the present time: how to give a great deal of knowledge to a system and how to use it?

ACKNOWLEDGMENTS

We wish to thank J. Pitrat, Research Director at the C.N.R.S., who leads our work in Artificial Intelligence for many years. We are indebted to him and to the members of his team for friendly and efficient aid.

We would also like to thank J. Arsac, Professor at the University P. and M. Curie (Paris VI) for many useful advices.

REFERENCES

1. Adam, A., Utilisation des Transformations Sémantiques pour la correction automatique des Programmes, Thèse de Doctorat d'Etat, Paris VI (1978).
2. Adam, A. and Laurent, J.P., Décomposition du graphe d'un programme permettant d'étudier la permutabilité des groupes d'instructions, *2ème Colloque International sur la Programmation, Paris* (April 1976).
3. Adam, A. and Laurent, J.P., Décomposition complète d'un graphe en fuseaux. Applications au graphe d'un programme, Annexe commun aux thèses d'état.
4. Arsac, J., Nolin, L., Ruggiu, G. and Vasseur, J.P., Le système de programmation structuré EXEL, *Rev. Tech. Thomson-CSF* **6** (3) (1974).
5. Arsac, J., *La construction de programmes structurés* (Dunod, Paris, 1977).
6. Ashcroft, E., Program proving without tears, *Colloq. IRIA, Arc et Senans* (Juillet 1975).
7. Bledsoe, W.W., Non resolution theorem proving, *Artificial Intelligence* **9** (1) (1977).
8. Brand, D., Analytic resolution in theorem proving, *Artificial Intelligence* 7 (4) (1976).
9. Boyer, R.S. and Moore, J.S., Proving theorems about LISP functions, *Proc. 3rd IJCAI* (1973).
10. Burstall, R.M. and Darlington, J.A., A system which automatically improves programs, *Acta Informatica* (1976).
11. Burstall, R.M. and Darlington, J.A., A transformation system for developing recursive programs, *J. ACM* **24** (1) (1977).
12. Chang, C.L., Lee, R.C.T. and Slagle, J.R., A direct method for verifying programs, Heuristic Laboratory Bethesda Maryland 20014.
13. Chang, C.L., A test oriented method for generating inductive assertions in program verification, IBM Research Laboratory, San José, CA 95193.
14. Dahl, O.J., Dijkstra, E. and Hoare, C.A.R., Structured programming, in: *APIC studies on data processing No. 8* (Academic Press, London, 1972).
15. Dijkstra, E., A constructive approach to the problem of program correctness, *BIT* **8** (1968).
16. Elspas, B., Green, M.G., Levitt, K.N. and Waldinger, R.J., Research in interactive program proving techniques, SRI, Meno Park, CA (May 1972).

17. Flavigny, B., Sur la détection a priori des erreurs dans les programmes, Thèse de 3ème cycle, Paris VI (December 1972).
18. Floyd, R.W., Assigning meanings to programs, in: *Proc. Symp. in Applied Mathematics, Vol. 19* (Am. Math. Soc. Providence, RI, 1967).
19. Floyd, R.W. and Knuth, D.E., Notes on avoiding GO TO statements. *Information Processing Lett.* **1** (1971).
20. Gerhardt, S.L., Knowledge about programs. A model and case study, *IEEE Conf. on Reliable Software* (1975).
21. Hearn, A.C., REDUCE II. A system and language for algorithm manipulations, *Proc. of 2nd Symp. on Symbolic and Algebraic Manipulations*, University of Utah (1971).
22. Hoare, C.A.R., An axiomatic basis for computer programming, *Comm. ACM* **12** (10) (1969).
23. Hoare, C.A.R., Proof of a program FIND, *Comm. ACM* **14** (1) (1971).
24. Katz, S.M. and Manna, Z., A heuristic approach to program verification, 3rd IJCAI (1973).
25. Katz, S.M. and Manna, Z., Logical analysis of programs, *Comm. ACM* **19** (4) (1976).
26. King, J., A program verifier, Thesis, Carnegie-Mellon University, Pittsburg (1969).
27. Knuth, D.E., Structured programming with GO TO statements, *ACM Computing Surveys* (December 1974).
28. Laurent, J.P., Un système qui met en évidence des erreurs sémantiques dans les programmes, Thèse de Doctorat d'Etat, Paris VI (1978).
29. Lee, R.C.T., Chang, C.L. and Waldinger, R.J., An improved program synthesis and its correctness, *Comm. ACM* **17** (4) (1974).
30. Loveman, D., Program improvement by source to source transformation, *J. ACM* **24** (1) (1977).
31. Malloy Brown, F., Doing arithmetic without diagrams, *Artificial Intelligence* **9** (2) (1977).
32. Martin, W.A. and Fateman, R.J., The MACSYMA system, *Proc. of 2nd Symp. on Symbolic and Algebraic Manipulation*, University of Utah (1971).
33. Manna, Z., The correctness of programs, *J. Comput. System. Sci.* **3** (2) (1969).
34. Manna, Z. and Pnueli, A., Axiomatic approach to total correctness of programs, Stanford Computer Science Dept. CS 73-382 (1973).
35. Manna, Z. and Waldinger, R.J., Knowledge and reasoning in program synthesis, *Artificial Intelligence* **6** (2) (1975).
36. Naur, P., Proof of algorithms by general snapshots, *BIT* **6** (1966).
37. Pastre, D., Automatic theorem proving in set theory, *Artificial Intelligence* **10** (1) (1978).
38. Ruth, G.R., Analysis of algorithm implementations, Thesis, MIT (May 1974).
39. Ruth, G.R., Intelligent program analysis, *Artificial Intelligence* **7** (1) (1976).
40. Standish, T.A., Harriman, D.C., Kibler, D.F. and Neighbors, J.M., The Irvine Program Transformation Catalogue, Dept. of Information and Computer Sciences, University of California, IRVINE (1976).
41. Waldinger, R.L. and Levitt, K.N., Reasoning about programs, *Artificial Intelligence* **5** (3) (1974).
42. Weigbreit, B., Complexity of synthesizing inductive assertions, *J. ACM* **24** (3) (1977).
43. Wertz, H., Un système de compréhension, d'amélioration et de correction de programmes incorrects, Thèse de 3ème cycle, Paris VI (1978).

Received 5 March 1979