Audris Kalnins
Dr. Comp. Sci.

Automation of testing, specification languages and CASE tools

Habilitation Thesis

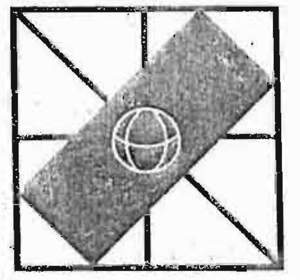
Collection of works

Part 1

Riga 1997

information processing 77

edited by
b. gilchrist



IFIP

north-holland

INFORMATION PROCESSING 77, B. GILCHRIST, EDITOR
© IFIP, NORTH-HOLLAND PUBLISHING COMPANY (1977)

AUTOMATIC CONSTRUCTION OF COMPLETE SAMPLE SYSTEM FOR PROGRAM TESTING

J. M. BARZDIN, J. J. BICEVSKIS and A. A. KALNINSH
Computing Centre of Latvian State University
Riga, USSR

A sample system is said to be complete for the given program if all executable branches (linear paths) of the program are executed on some sample of this system. The algorithmic solvability of the construction of a finite complete sample system is proven for a sufficiently wide class of programs. In conclusion the first results of the implementation of such a system are presented.

1. INTRODUCTION

Considerable success in precise methods of program correctness testing has been achieved in recent years (J.McCarthy, D.Scott, R.Floyd, C.R.Hoare, S.Igarashi et al.). In spite of this in practice the most widely used testing method is the old one in which some sample system is constructed and then the program is run on this sample. The sample system choice is the most sophisticated part of such a testing process. Usually the programmer tends to find a sample system such that every branch of the program is executed when running the program on an appropriate sample of the system. Such a sample system we shall call complete. If the program runs correctly on such a system the programmer (and the customer too) usually considers the program to be correct. This heuristic principle is widely and successfully used in practice. It is considered also in[1]. We shall not investigate the theoretical foundations of this principle but assume it as basic. Thus the main problem in the automation of program testing is the automatic construction of a complete sample system for a given program. It is clear that the problem of the construction of a complete sample system is algorithmically unsolvable in general (see also Theorems 2,3 below). The authors have shown [2 - 4] that for a sufficiently wide class of data processing programs this problem is solvable. The aim of this paper is to present these together with some further results and also give the first experience of the experimental implementation of the above mentioned system. Let us note that some papers related to these topics [6,7,8 et al.] have appeared recently. However, the authors of these papers confine themselves mainly to the analysis of paths specified by the user.

2. A SOLVABLE CASE

In order to expose the principal ideas we introduce a very simple programming language for the processing of sequentional files. Nevertheless, a large part of business data processing problems can be formalized in this language (adequately enough to investigate the construction of a complete sample system). This language can be characterized by the fact that the values taking part in the comparison instructions are undeformed (i.e., such as read from input). This restriction is acceptable in practice because it is typical for data processing problems, that program logic is controlled only by input data (e.g., record type) and that these data are used in comparison instructions undeformed.

Now let us describe this language. Let file be a variable whose values are finite sequences of integers $(X_1, X_2, \ldots, X_p)$, where X_i is i-th record of file. Each program has a finite set of input files and a finite set of output files. The program processes the values of input files into values of output files. The program has also inner variables with integer values (the initial values are set to 0). Let X be an input file, Y - an output file, t,u - inner variables and c - a constant (a fixed integer). The following instructions are available.

1. $X \Rightarrow t$. The current record of file X is assigned to variable t. Thus if $X=(X_1,X_2, \ldots, X_p)$, the first occurence of instruction $X \Rightarrow t$ assigns the value X_1 to t, the second - X_2 and so on. The instruction has two exits: "+" if the current record exists and exit "-" if the file is exhausted. In the last case the value of t is not changed (Input instruction).

2. $t \Rightarrow Y$. The value of variable t is assigned to the current record of file Y (Print instruction).

3. $u \Rightarrow t$ (respectively, $c \Rightarrow t$). The value of variable u (constant c) is assigned to variable t (Assignment instruction).

4. $u < t$ (respectively, $c < t$, $u < c$). The instruction has two exits: if the value of u (respectively c) is less than the value of t (respectively c) then the exit "+" is used, otherwise.- the exit "-" (Comparison instruction).

5. NOP. Dummy instruction (nothing is done). It is used instead of instructions not essential for the construction of a complete sample system when more general programming languages are reduced to this one. (Informally, these are nonconditional instructions not affecting the range of variable values used in comparisons).

6. STOP

Let L_o be the language generated by the instructions 1 - 6, where the programs are given as flowcharts over this instruction set. We shall call L_o the base language. Fig. 1 gives an example of a program, which creates a new sorted file Y by merging sorted files A and B. The program has a bug: control from instruction 7 is passed to instruction 12 (instead of 10).

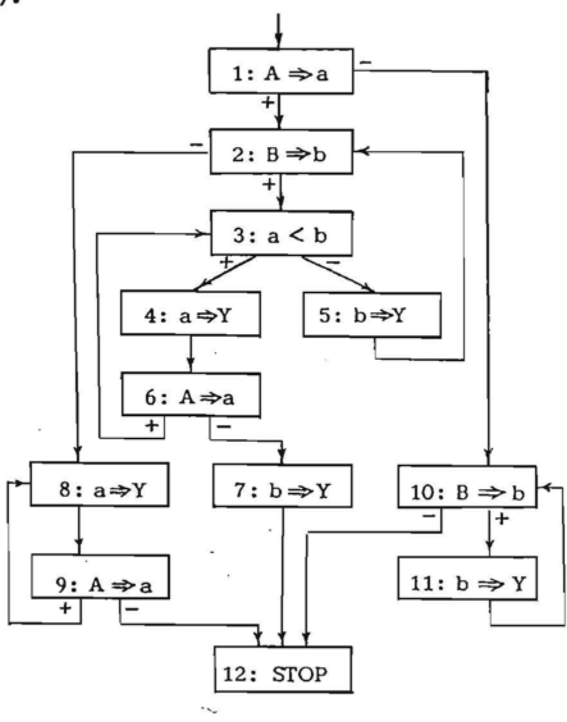


Fig. 1

By a branch of the program we understand a linear path between two adjacent conditional instructions (i.e., instructions with two exits). For example, the program in Fig. 1 has branches (1:A ⇒a+), (10:B⇒ b+, 11:b ⇒Y), (1:A ⇒ a-) etc.

Let the program have input files A, B, ... By a sample we shall understand fixed values of all these files: $A=A^o$, $B=B^o$, ... Let us say that sample P realizes the branch h of program T if this branch is executed while running program T on sample P. E.g., when the program in Fig. 1 is run on the sample A=(0), B=(1), the path (1:A ⇒a+, 2:B ⇒b+, 3:a < b+, 4:a ⇒Y, 6:A ⇒a-, 7:b ⇒Y, 12:STOP) containing branches (1:A ⇒ a+), (2:B ⇒b+), ... is executed. The sample system is said to be a complete sample system (CSS) for the given program, if every branch realizable by some arbitrary sample is realized by some sample in this system. For the program in Fig. 1, for example, the following sample system is complete $P_1 = \{A=(0,1), B=(2)\}$, $P_2 = \{A=(0), B=(1,2,3)\}$, $P_3 = \{A=(2), B=(0,2)\}$, $P_4 = \{A=(1,2,3), B=(0)\}$, $P_5 = \{A=(\ ), B=(0,1,2)\}$. It is easy to see that the bug in the program is found in this system. Evidently, for every program there exists a finite CSS. The main problem is to find this system.

THEOREM 1. There is an algorithm for constructing a finite complete sample system for every program in L_o.

The proof will consist of several auxiliary assertions. Let T be a program in L_o and $\alpha = (K_1, K_2, \ldots, K_r)$ a path in T, K_i - an instruction with fixed exit (+ or -). The exit of instruction K_i must enter K_{i+1}. The program in Fig. 1 contains, for example, the path α=(1:A ⇒ a+, 2:B ⇒ b+, 3:a < b+, 4:a ⇒Y, 6:A ⇒ a+) (or simply α = (1+,2+,3+,4,6+), if only labels of instructions are used). If it is not stated otherwise, we shall assume that the path always begins with the first instruction of the program. The following system of inequalities $N(\alpha)$ is related to path α: $N(\alpha) = \{M_o, M_1, \ldots, M_r\}$, where $M_o = \{t_o = 0, u_o = 0 \ldots\}$ t, u - inner variables, and M_i, i=1, ..., r, is the subsystem corresponding to instruction K_i in the following way. Let X be an input file and t, u inner variables in the program. Let t_k, u_l be variables denoting the values of variables t, u after the execution of path $\alpha_{i-1} = (K_1, \ldots, K_{i-1})$ and X_s the last record of file X read during it (at the beginning corresponding variables are t_o, u_o, X_o). Let c be a constant. System M_i is defined in the following manner:

(1) If K_i = (X ⇒u-), then $M_i = \{X < 0\}$. By inequality $X < 0$ we code the exhaustion of file X.

(2) If K_i = (X ⇒u+) and if $M_1, M_2, \ldots, M_{i-1}$ do not contain inequality $X<0$ (i.e. no instruction of type X⇒ ...- has been performed) then $M_i = \{u_{l+1} = X_{s+1}\}$. In this case new variables u_{l+1} and X_{s+1} are introduced which have the same sense for instruction K_{i+1} as u_l and X_s have for K_i. If inequality $X<0$ has occured already, then $M_i = \{X<0, X>0\}$, i.e., a contradictory system is chosen.

(3) If K_i = (t ⇒u) (or K_i = (c ⇒u)), then $M_i = \{u_{l+1} = t_k\}$ (or $M_i = \{u_{l+1} = c\}$). A new variable u_{l+1} is again introduced in this case.

(4) If K_i = (t < u+) (or K_i = (c < u+) or K_i = (t < c+)) then $M_i = \{t_k + 1 \leq u_l\}$ (or $M_i = \{c+1 \leq u_l\}$, or $M_i = \{t_k + 1 \leq c\}$).

(5) If K_i = (t<u-) (or K_i = (c<u-), or K_i = (t<c-)) then $M_i = \{t_k \geq u_l\}$ (or $M_i = \{c \geq u_l\}$, or $M_i = \{t_k \geq c\}$).

Let us give an example. For α' = (1+,2+,3+) we have $N(\alpha') = \{a_o = 0, b_o = 0, a_1 = A_1, b_1 = B_1, a_1 + 1 \leq b_1\}$. From the construction of $N(\alpha)$ there follows:

LEMMA 1. The path α is realizable iff the system $N(\alpha)$ has an integer solution. Any solution of $N(\alpha)$ with respect to variables-records of input files yields a sample realizing path α.

Our aim is to reduce $N(\alpha)$ while preserving the existence (or the nonexistence) of the solution in such a way that there will be only a finite number of possible reduced systems for the given program. Let the program have input files A, B, ... and inner variables t, u, ... Then the system $N(\alpha)$ contains, in general, variables A, B, ..., $A_1, A_2, \ldots, A_d$, $B_1, B_2, \ldots, B_e, \ldots \ldots, t_1, t_2, \ldots, t_f, u_1, u_2, \ldots, u_g, \ldots \ldots$. Let us remember that inner variables with maximal subscripts - t_f, u_g, ... denote values of inner variables t, u, ... after the execution of path α. These variables t_f, u_g, ..., as well as variables A, B, ... denoting input files - we shall call active variables of system $N(\alpha)$. For example, the system $N(\alpha')$ from the previous example has active variables a_1, b_1. Out of all the variables of $N(\alpha)$, only the active ones can enter inequalities added to $N(\alpha)$ during the construction of $N(\alpha+\beta)$. Now let us define the exclusion of variable y from the system of inequalities N, taking into account all the inequalities induced by transitivity.

Formally, we consider all the pairs x, z of variables and/or constants different from y, for which there exist inequalities $x + p_1 \leqslant y$ and $y + p_2 \leqslant z$ ($p_1 \geqslant 0$, $p_2 \geqslant 0$; equality x'=y' is substituted here by inequalities $x' \leqslant y'$ and $y' \leqslant x'$). For each of these pairs we add inequality $x + (p_1 + p_2) \leqslant z$ to the system N. It turns out that it is possible to restrict the range of the constant appearing in the added inequalities preserving again the existence of a solution: if $p_1 + p_2 > C_0 = C_2 - C_1$, where C_2 is the maximum and C_1 the minimum constants of the program, then $p_1 + p_2$ is changed to $C_0 + 1$ in the added inequality. Then we delete all inequalities and consequently, equalities, containing y from N. If N contains inequalities of type $y + p \leqslant y$ with $p > 0$ then these inequalities are substituted by some standard contradicting inequality, e.g., $1 < 0$ (because the new system must have no solutions). Now let us exclude, one after another, all inactive variables from $N(\alpha)$. Thus, we get a new system of inequalities. Then we drop all the subscripts of the variables in $\mathfrak{V}$. The resulting system is denoted by $S(\alpha)$ and called a state. Informally the state describes relations between current values of inner variables. In the previous example state $S(\alpha') = \{a+1 < b\}$ corresponds to the path $\alpha' = (1+, 2+, 3+)$.

LEMMA 2. Path α is realizable (i.e., system $N(\alpha)$ has an integer solution) iff the state $S(\alpha)$ is consistent (i.e., $S(\alpha)$ has an integer solution as a system of inequalities).

It follows from the definition of state that if two paths α and β end with the same instruction and γ' is a continuation of both paths then $S(\alpha) = S(\beta)$ implies $S(\alpha + \gamma') = S(\beta + \gamma')$. Hence from Lemma 2 there follows:

LEMMA 3. Let paths α and β end with one and the same instruction and $S(\alpha) = S(\beta)$. Let γ' be a continuation of these paths. Then path $\alpha + \gamma'$ is realizable iff path $\beta + \gamma'$ is realizable.

Let us note that state $S(\alpha)$ can be effectively constructed from path α and the number of different states is finite for a given program. Lemma 3 gives us the possibility (using a technique popular in automata theory) to construct effectively from the program a finite system of realizable paths

$$\alpha^1, \alpha^2, \ldots, \alpha^p$$

such that every realizable branch is contained by some of these paths. Now the solutions of corresponding systems of inequalities $N(\alpha^1), N(\alpha^2), \ldots, N(\alpha^p)$ yields us a sample system $P_1, P_2, \ldots, P_p$ which is a complete sample system. This concludes the proof of Theorem 1.

3. UNSOLVABLE CASES

Let us consider a language L_1 where the same file X can be opened (i.e., read from the first record) repeatedly. Formally, L_1 is obtained from the base language L_0 by adding instructions of the type REOPEN X.

THEOREM 2. There exists no algorithm for constructing a finite complete sample system for every program in L_1 (a subclass of programs in L_1 with two input files with one usage of REOPEN for each of them is sufficient for non-existence).

We consider two-tape automata by Rabin and Scott [9]. These automata may be represented by programs in base language L_0 with two input files. According to [9], the following problem is undecidable : determine for two-tape automata A and B the emptiness of the intersection of languages L_A and L_B represented by these automata. We identify tapes of automata with two input files. It is easy to construct a program P_{AB} using the REOPEN instruction only once for each of the files where the STOP instruction is accessible iff $L_A \cap L_B \neq \emptyset$. Hence it follows that the emptiness of $L_A \cap L_B$ can be decided by means of a complete sample system.

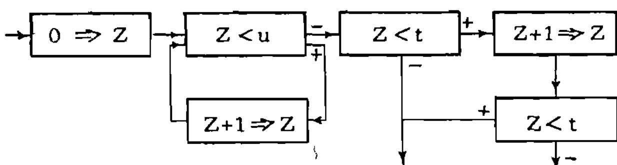
Now let us consider a language L_2 which is obtained from the base language by adding variables of a new type - counters and the following instructions for them:

$$C \Rightarrow Z, \quad Z + 1 \Rightarrow Z, \quad Z < t$$

where Z is a counter and t - an inner variable. Instruction $Z < t$ allows us to compare counters with records of input files in L_2.

THEOREM 3. There exists no algorithm for constructing a complete sample system for every program in L_2.

The proof of Theorem 3 relies on testing, by means of constructions of language L_2, whether or not the input file is a sequence of integers which is the configuration sequence of some Minsky machine. [10] It is easy to see that this test can be reduced to the tests of type "is it t=u+1?". Such a test (for $u \geqslant 0$) can be performed by means of one counter Z:



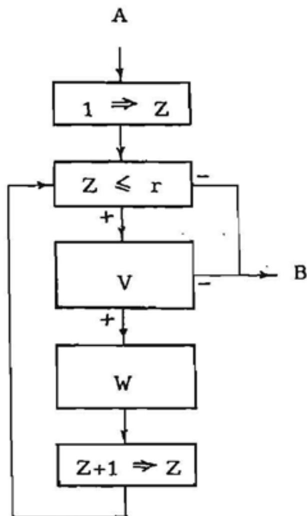
At the same time, if counters can be compared only with constants, the problem of construction of CSS is solvable. This follows from the same ideas as the proof of Theorem 1. Only the values of counters if they lie between the minimum and maximum constants of the program are included in the state.

4. SOLVABLE EXTENSIONS

Theorem 3 shows that the unlimited usage of counters in a program causes the unsolvability of the problem of CSS construction. However, in practice counters are mainly used for loop organization. This is done by means of the DO statement:

DO Z=1 TO r WHILE V; W; END;

where W - the body of the loop is a program block (by program block we understand part of the program consisting of base instructions and, possibly, DO statements and having a single entry and a single exit), V is a boolean expression constructed from comparisons of L_0 (e.g., $(t<u)$ & $(5<t)$), and r - the bound of the loop is an inner variable. A DO statement (called also a DO-loop) will be interpreted as an abreviation of the following program block:



It is assumed that counter Z is used in no instructions other than the above mentioned ones $Z \leq r$ and $Z+1 \Rightarrow Z$ used for loop organization.

Let us consider the programming language generated by the base instruction set and the DO statement. There is no algorithm for constructing a CSS for every program in this language (a stronger version of Theorem 3). The proof is close to the one used for Theorem 3 except that a slightly different coding of the Minsky machine configurations is used. This proof of unsolvability strongly relies on comparing the loop bound r with other inner variables. Now let us exclude this possibility.

We shall not allow the use of the loop bound r in comparisons with other inner variables and in assignments. This means that the loop bound r along with the loop organization instruction $Z \leq r$ can be used only in input instructions ($X \Rightarrow r$), comparisons with constants ($r < c$, $c < r$) and print instructions ($r \Rightarrow Y$). In practice these restrictions are not essential but they usually hold for real programs. Let us note that several DO-loops can have a common bound r. The programming language generated by the base instruction set and the DO statement with above mentioned restrictions is called L_3.

THEOREM 4. There exists an algorithm for constructing a finite complete sample system for every program in L_3.

A detailed proof of Theorem 4 is rather lengthy, so we shall outline only the main ideas used. By a simple state we shall understand a state in the sense of Theorem 1, i.e., one obtained by ignoring the instructions containing counters and loop bound. Let us consider a DO-loop having no nested DO-loops in it. By entering the DO-loop in a simple state S (at entry point A) and going through all the possible values of bound r we can obtain, at the exit of the loop (point B), generally speaking, distinct simple states $S_1, S_2, \ldots, S_a$. Further, for every state S_i there exists the set R_i of the values of bound r, for which the state S_i is reached at the exit. More precisely, $\bar{r} \in R_i$ iff for $r = \bar{r}$ and state S at point A there exists a realizable path through the DO-loop beginning in the point A and reaching point B in the state S_i. The set R is said to be regular if there exists a regular expression $\bar{R}$ in the binary alphabet $\{1,0\}$ such that for $r \geq 0$

$$r \in R \text{ iff } \underbrace{11\ldots1}_{r} \in \bar{R}$$

and for $r < 0$

$$r \in R \text{ iff } \underbrace{00\ldots0}_{|r|} \in \bar{R}.$$

The expression $\bar{R}$ is said to be regular representation of the set R. Regular expressions are preferable due to the decidability of the emptiness problem.

LEMMA 1. Set R_i is regular for every i. States $S_1, \ldots, S_a$ and the corresponding regular representations of sets $R_1, \ldots, R_a$ can be constructed effectively from the DO-loop and state S.

Theorem 4 can be proved by Lemma 1 in the simplest case when the program contains only non-nested DO-loops, none of which includes instructions involving bounds of other DO-loops. In the general case some generalization of Lemma 1 is necessary.

Let us order the variables used as loop bounds in the program:

$$r^{(1)}, r^{(2)}, \ldots, r^{(k)}.$$

Let us consider a set of strings of the type

$$\langle r_I^{(1)}, r_{II}^{(1)}, r_I^{(2)}, r_{II}^{(2)}, \ldots, r_I^{(k)}, r_{II}^{(k)} \rangle,$$

where $r_I^{(j)} \in M$, $r_{II}^{(j)} \in M \cup \{*\}$, M is the set of integers and * a special symbol.

The set of strings is said to be regular if it can be expressed as a finite union of cartesian products of regular sets ($\{*\}$ is considered to be a regular set):

$$R_{I,1}^{(1)} \times R_{II,1}^{(1)} \times \ldots \times R_{I,1}^{(k)} \times R_{II,1}^{(k)} \cup \ldots \cup R_{I,b}^{(1)} \times R_{II,b}^{(1)} \times \ldots \times R_{I,b}^{(k)} \times R_{II,b}^{(k)},$$

$R_{I,1}^{(1)}, R_{II,1}^{(1)}, \ldots, R_{I,b}^{(k)}, R_{II,b}^{(k)}$ – regular sets.

The expression

$$\bar{R}_{I,1}^{(1)} \times \bar{R}_{II,1}^{(1)} \times \ldots \times \bar{R}_{I,1}^{(k)} \times \bar{R}_{II,1}^{(k)} \cup \ldots \cup \bar{R}_{I,b}^{(1)} \times \bar{R}_{II,b}^{(1)} \times \ldots \times \bar{R}_{I,b}^{(k)} \times \bar{R}_{II,b}^{(k)},$$

where

$$\bar{R}_{I,1}^{(1)}, \bar{R}_{II,1}^{(1)}, \ldots, \bar{R}_{I,b}^{(k)}, \bar{R}_{II,b}^{(k)}$$

are regular representations of the sets

$$R_{I,1}^{(1)}, R_{II,1}^{(1)}, \ldots, R_{I,b}^{(k)}, R_{II,b}^{(k)},$$

is said to be a regular representation of the corresponding set of strings.

Let a program block, with entry C and exit D, be given. Let S be a simple state at point C and S_i be a simple state accesible at exit D. Let us denote by U_i the following set of strings.

$\langle r_{I}^{(1)}, r_{II}^{(1)}, \dots, r_{I}^{(k)}, r_{II}^{(k)} \rangle \in U_i$
iff for $r^{(1)} = r_{I}^{(1)}, \dots, r^{(k)} = r_{I}^{(k)}$

and state S at point C there exists a realizable path through the block, beginning at point C, reaching point D in state S_i, and satisfying the condition:

if $r_{II}^{(j)} = *$, then the path contains no input instruction of the type $\langle file \rangle \Rightarrow r^{(j)}$;

if $r_{II}^{(j)}$ is a number, then the path contains one or several input instructions $\langle file \rangle \Rightarrow r^{(j)}$ and $r_{II}^{(j)}$ is a possible value of variable $r^{(j)}$ at point D on the given path ($j=1, \dots, k$).

LEMMA 2. The set of strings U_i is regular for every i. The possible states $S_1, \dots, S_d$ at the exit of the block and the corresponding regular representations of the sets $U_1, \dots, U_d$ can be constructed effectively from the program block and state S.

The lemma is proved by induction over the depth of nesting of loops in the block. For depth 1 Lemma 2 is a slight strengthening of Lemma 1.

Now let us consider a block path $\alpha=(K_1, K_2, \dots, K_m)$. It differs from the usual path in that the K_i can be either a base instruction or a DO-loop. If K_i is a DO-loop we fix one of the possible simple states S_i at its exit. An instance of a block path is $\alpha=(X \Rightarrow a+, a<e-, (D,S_i), a \Rightarrow Y)$, where D is some DO-loop. Now let us define the total state $Z(\alpha)$ as a pair $(S(\alpha), W(\alpha))$, where $S(\alpha)$ is a simple state and $W(\alpha)$ is a regular expression describing all the possible strings $\langle \bar{r}^{(1)}, \dots, \bar{r}^{(k)} \rangle$ of numbers acceptable as the values of the variables $r^{(1)}, \dots, r^{(k)}$ at the end of path α. $W(\alpha)$ can be easily constructed using Lemma 2. It follows from the construction that for a given program the number of distinct total states is finite. Moreover, an analogue of Lemma 3 from Theorem 1 holds for the total states. Hence arguments analogous to those used in proving Theorem 1 lead to a proof of Theorem 4.

Now some words about another extension of the base language L_0. So far we have considered only sequential files. The authors have investigated also the case of direct access files in [5] and have shown that the CSS construction problem is solvable for a sufficiently wide class of such programs.

5. PRACTICAL IMPLEMENTATION

Some improvements of the above mentioned algorithm for constructing a CSS are important in practical implementation.

1. First, a set of essential instructions of the program is selected, i.e., a set of possibly fewer instructions, containing the first instruction, and, at least, one instruction from every loop. Then, during the construction of realizable paths $\alpha^1, \alpha^2, \dots, \alpha^p$, states are attached only to essential instructions.

2. Next, a set of essential inner variables is selected for each essential instruction. The variable is essential, if its value is used (directly or through assignments) in further comparison instructions. The improvement now consists in the following. When constructing the state $S(\alpha)$ from $N(\alpha)$, we exclude not only the inactive variables but also active variables t_f where the corresponding inner variable t is not essential for the instruction entered by path α (more precisely the instruction entered by the exit of the last instruction of α). This can greatly reduce, in practice, the number of states to be considered. At the same time analogues of Lemma 2 and 3 are preserved.

3. During the construction of realizable paths $\alpha^1, \alpha^2, \dots, \alpha^p$, yielding a CSS, the following principle is used: the branch, contained the least number of times in the paths $\alpha^1, \alpha^2, \dots, \alpha^j$ already constructed, is chosen as the next branch for analysis. This principle frequently allows to construct a CSS using only part of possible states.

The following improvement of algorithm allows to widen the acceptable instruction repertoire. Add/subtract instructions are allowed in the case when either the result or at least one of the operands is not involved in comparisons. In this case the "free" value can be chosen arbitrarily.

An experimental implementation of a CSS construction system was accomplished for a COBOL-like language. No more that 120K Bytes of main storage were used with a time limitation of 10 minutes per program at CPU speed of approximately 40000 operation/sec. In this environment, the system was able to construct a CSS for approximately 70% of data processing programs containing no more than 300 statements. The results obtained make us believe in the possibility of implementing an automatic CSS construction system, having a speed comparable to that of high level language compilers and applying to most real programs.

REFERENCES

[1] E.F.Miller and M.R.Paige, Automatic generation of software testcases, Eurocomp Conference Proceedings, 1974, 1-12.

[2] J.M.Barzdin, J.J.Bicevskis and A.A.Kalninsh, Construction of complete sample system for program testing, Uchonye zapiski Latviiskogo gosudarstvennogo universiteta, vol.210, 1974, 152-187 (Russ.)

[3] J.M.Barzdin, J.J.Bicevskis and A.A.Kalninsh, A solvable and unsolvable cases of the problem of construction of a complete sample system, Uchonye zapiski Latviiskogo gosudarstvennogo universiteta, vol.210, 1974, 188-205 (Russ.)

[4] J.M.Barzdin, J.J.Bicevskis and A.A.Kalninsh, Construction of complete sample system for correctness testing, Lecture Notes in Computer Science, vol.32, Springer-Verlag, Berlin, 1975, 1-12.

[5] J.M.Barzdin and A.A.Kalninsh, Construction of complete sample system for programs using direct access files, Uchonye zapiski Latviiskogo gosudarstvennogo universiteta, vol.233, 1975, 123-154 (Russ.)

[6] W.E.Howden, Methadology for the generation of Program test data, IEEE Transactions on Computers, vol.C-24,No.5,May 1975,554-559.
[7] J.C.King, A new approach to program testing, Proc.Int.Conf.Reliable Sofware, Apr.1975,228-233.
[8] L.A.Clarke, A system to generate test data and symbolically execute programs, IEEE Transactions on Software Engineering, vol.SE-2, No.3, Sept. 1976,215-222.
[9] M.O.Rabin and D.Scott, Finite automata and their decision problems, IBM J.of Research and Development, vol.3,No.2,1959,114-125.
[10] M.L.Minsky, Finite and infinite machines, Prentice-Hall, Englewood Cliffs,N.Y.,1967.

Lecture Notes in Computer Science

Vol. 1: GI-Gesellschaft für Informatik e.V. 3. Jahrestagung, Hamburg, 8.–10. Oktober 1973. Herausgegeben im Auftrag der Gesellschaft für Informatik von W. Brauer. XI, 508 Seiten. 1973. DM 32,–

Vol. 2: GI-Gesellschaft für Informatik e.V. 1. Fachtagung über Automatentheorie und Formale Sprachen, Bonn, 9.–12. Juli 1973. Herausgegeben im Auftrag der Gesellschaft für Informatik von K.-H. Böhling und K. Indermark. VII, 322 Seiten. 1973. DM 26,–

Vol. 3: 5th Conference on Optimization Techniques, Part I. (Series: I.F.I.P. TC7 Optimization Conferences.) Edited by R. Conti and A. Ruberti. XIII, 565 pages. 1973. DM 38,–

Vol. 4: 5th Conference on Optimization Techniques, Part II. (Series: I.F.I.P. TC7 Optimization Conferences.) Edited by R. Conti and A. Ruberti. XIII, 389 pages. 1973. DM 28,–

Vol. 5: International Symposium on Theoretical Programming. Edited by A. Ershov and V. A. Nepomniaschy. VI, 407 pages. 1974. DM 30,–

Vol. 6: B. T. Smith, J. M. Boyle, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler, Matrix Eigensystem Routines – EISPACK Guide. X, 387 pages. 1974. DM 28,–

Vol. 7: 3. Fachtagung über Programmiersprachen, Kiel, 5.–7. März 1974. Herausgegeben von B. Schlender und W. Frielinghaus. VI, 225 Seiten. 1974. DM 20,–

Vol. 8: GI-NTG Fachtagung über Struktur und Betrieb von Rechensystemen, Braunschweig, 20.–22. März 1974. Herausgegeben im Auftrag der GI und der NTG von H.-O. Leilich. VI, 340 Seiten. 1974. DM 26,–

Vol. 9: GI-BIFOA Internationale Fachtagung: Informationszentren in Wirtschaft und Verwaltung. Köln, 17./18. Sept. 1973. Herausgegeben im Auftrag der GI und dem BIFOA von P. Schmitz. VI, 259 Seiten. 1974. DM 22,–

Vol. 10: Computing Methods in Applied Sciences and Engineering, Part 1. International Symposium, Versailles, December 17–21, 1973. Edited by R. Glowinski and J. L. Lions. X, 497 pages. 1974. DM 34,–

Vol. 11: Computing Methods in Applied Sciences and Engineering, Part 2. International Symposium, Versailles, December 17–21, 1973. Edited by R. Glowinski and J. L. Lions. X, 434 pages. 1974. DM 30,–

Vol. 12: GFK-GI-GMR Fachtagung Prozessrechner 1974. Karlsruhe, 10.–11. Juni 1974. Herausgegeben von G. Krüger und R. Friehmelt. XI, 620 Seiten. 1974. DM 42,–

Vol. 13: Rechnerstrukturen und Betriebsprogrammierung, Erlangen, 1970. (GI-Gesellschaft für Informatik e.V.) Herausgegeben von W. Händler und P. P. Spies. VII, 333 Seiten. 1974. DM 30,–

Vol. 14: Automata, Languages and Programming – 2nd Colloquium, University of Saarbrücken, July 29–August 2, 1974. Edited by J. Loeckx. VIII, 611 pages. 1974. DM 48,–

Vol. 15: L Systems. Edited by A. Salomaa and G. Rozenberg. VI, 338 pages. 1974. DM 30,–

Vol. 16: Operating Systems, International Symposium, Rocquencourt 1974. Edited by E. Gelenbe and C. Kaiser. VIII, 310 pages. 1974. DM 30,–

Vol. 17: Rechner-Gestützter Unterricht RGU '74, Fachtagung, Hamburg, 12.–14. August 1974, ACU-Arbeitskreis Computer-Unterstützter Unterricht. Herausgegeben im Auftrag der GI von K. Brunnstein, K. Haefner und W. Händler. X, 417 Seiten. 1974. DM 35,–

Vol. 18: K. Jensen and N. Wirth, PASCAL – User Manual and Report. VI, 167 pages. 2nd Edition 1975. DM 20,–

Vol. 19: Programming Symposium. Proceeding, Colloque sur la Programmation, Paris, April 9–11, 1974. V, 425 pages. 1974. DM 35,–

Vol. 20: J. Engelfriet, Simple Program Schemes and Formal Languages. VII, 254 pages. 1974. DM 25,–

Vol. 21: Compiler Construction, An Advanced Course. Edited by F. L. Bauer and J. Eickel. XIV. 621 pages. 1974. DM 42,–

Vol. 22: Formal Aspects of Cognitive Processes. Proceedings, Interdisciplinary Conference, Ann Arbor, March 1972. Edited by T. Storer and D. Winter. V, 214 pages. 1975. DM 23,–

Vol. 23: Programming Methodology. 4th Informatik Symposium, IBM Germany Wildbad, September 25–27, 1974. Edited by C. E. Hackl. VI, 501 pages. 1975. DM 39,–

Vol. 24: Parallel Processing. Proceedings of the Sagamore Computer Conference, August 20–23, 1974. Edited by T. Feng. VI, 433 pages. 1975. DM 35,–

Vol. 25: Category Theory Applied to Computation and Control. Proceedings of the First International Symposium, San Francisco, February 25–26, 1974. Edited by E. G. Manes. X, 245 pages. 1975. DM 25,–

Vol. 26: GI-4. Jahrestagung, Berlin, 9.–12. Oktober 1974. Herausgegeben im Auftrag der GI von D. Siefkes. IX, 748 Seiten. 1975. DM 49,–

Vol. 27: Optimization Techniques. IFIP Technical Conference. Novosibirsk, July 1–7, 1974. (Series: I.F.I.P. TC7 Optimization Conferences.) Edited by G. I. Marchuk. VIII, 507 pages. 1975. DM 39,–

Vol. 28: Mathematical Foundations of Computer Science. 3rd Symposium at Jadwisin near Warsaw, June 17–22, 1974. Edited by A Blikle. VIII, 484 pages. 1975. DM 37,–

Vol. 29: Interval Mathematics. Proceedings of the International Symposium Karlsruhe, West Germany, May 20–24, 1975. Edited by K. Nickel. VI, 331 pages. 1975. DM 30,–

Vol. 30: Software Engineering. An Advanced Course. Edited by F. L. Bauer. (Formerly published 1973 as Lecture Notes in Economics and Mathematical Systems, Vol. 81) XII, 545 pages. 1975. DM 42,–

Vol. 31: S. H. Fuller, Anaoysis of Drum and Disk Storage Units. IX, 283 pages. 1975. DM 28,–

Vol. 32: Mathematical Foundations of Computer Science 1975. Proceedings 1975. Edited by J. Bečvář. X, 476 pages. 1975. DM 39,–

Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

32

Mathematical Foundations of Computer Science 1975

4th Symposium, Mariánské Lázně, September 1-5, 1975

Edited by J. Bečvář

Springer-Verlag
Berlin · Heidelberg · New York 1975

CONSTRUCTION OF COMPLETE SAMPLE SYSTEM FOR CORRECTNESS TESTING

J.M.Barzdin, J.J.Bičevskis, A.A.Kalninsh
Computing Center of Latvian State University
Riga, USSR

INTRODUCTION

In spite of success in axiomatic methods for program correctness proofs (J.McCarthy, D.Scott, R.Milner, C.Hoare e.a.) the old well-known method for correctness testing (debugging) of programs strongly prevails. According to this method a set of samples is constructed and program is run on these samples. If the program yields correct results on these samples, programmer usually believes his program being correct. The choice of a suitable sample set is the most sophisticated part of debugging process. Usually programmer tends to find a set of samples such that every branch of program is executed when running the program on an appropriate sample of this set. If program runs correctly on all the samples of this set which is called complete sample system henceforth the programmer has great certainity that his program will run correctly on every input. Of course, this criterion is not absolute, nevertheless it is widely and successfully used in practice.

Thus the main problem in the automation of debugging process is automatic construction of complete sample system for a given program.

It is clear that the problem of constructing complete sample system is algorithmically unsolvable in general (see also Theorems 2,3,4 below). The aim of this paper is to show that for a sufficiently wide class of data processing programs this problem is solvable.

Some of the results given here can be found with full proofs in [1] , [2] by authors.

SOLVABLE AND UNSOLVABLE CASES

Now let us define a programming language for file processing using sequential access method. In this language a great part of data processing problems can be formalized (adequately enough to investigate construction of complete sample system).

Now let file be a variable whose values are finite sequences of integers $(n_1, n_2, \ldots, n_r)$, n_i being i-th record of file. Let parameter be a variable with integer values.

Each program has a finite set of input files and input parameters. The program has also a finite set of output files. The program processes the values of input files and parameters creating the values of output files.

Program has also inner variables with integer values (the initial values being equal to 0). Two types of inner variables are available - main variables and counters.

Let X be an input file, Y - an output file, n - an input parameter, t,u - main inner variables, z - a counter and c - a constant (a fixed integer).

The following instructions are available.

1. $X \Rightarrow t$. The current record of file X is assigned to variable t. Thus if $X=(n_1,n_2,\ldots,n_r)$, the first occurence of instruction $X \Rightarrow t$ assigns the value n_1 to t, the second - n_2 and so on. The instruction is conditional. It has two exits: the exit "+" when current record exists and exit "-" when end of file is reached. In the last case the value of t is not changed.

2. $t \Rightarrow Y$. The value of variable t is assigned to the current record of file Y.

3. $a \Rightarrow t$ where $a \in \{u,n,c\}$. The value of a is assigned to variable t.

4. $b \Rightarrow z$, where $b \in \{n,c\}$. The value of b is assigned to counter z.

5. $z+1 \Rightarrow z$. The semantics is evident.

6. $a<b$, where $a \in \{u,n,c\}$, $b \in \{t,n,c\}$. This conditional instruction has two exits: if value of a is less than value of b, the exit "+" is used, otherwise - the exit "-".

7. $z<b$ (or $b<z$), where $b \in \{n,c\}$. Semantics is analogic to the previous case.

8. NOP. Dummy instruction (nothing is done). It is used instead of instructions not essential for construction of complete sample system when more general programming languages are reduced to this

one. (Informally, these are nonconditional instructions not affecting the range of variable values used in conditions).

9. STOP.

Let L_o be the language generated by the instructions 1-9. Let L_* be the language (called base language) which is obtained from L_o by omitting counters (i.e., instructions 4,5,7).

Programs will be given as flowcharts over an instruction set. The instructions have labels for reference.

Fig.1 gives an example of program in L_o, which creates a new sorted file Y by merging sorted files A and B. Y contains first m records from A and n first records from B. m and n are input parameters, A and B - input files, Y - output file. Program has a bug: control from instruction 12 is passed to the instruction 5 (instead of 4).

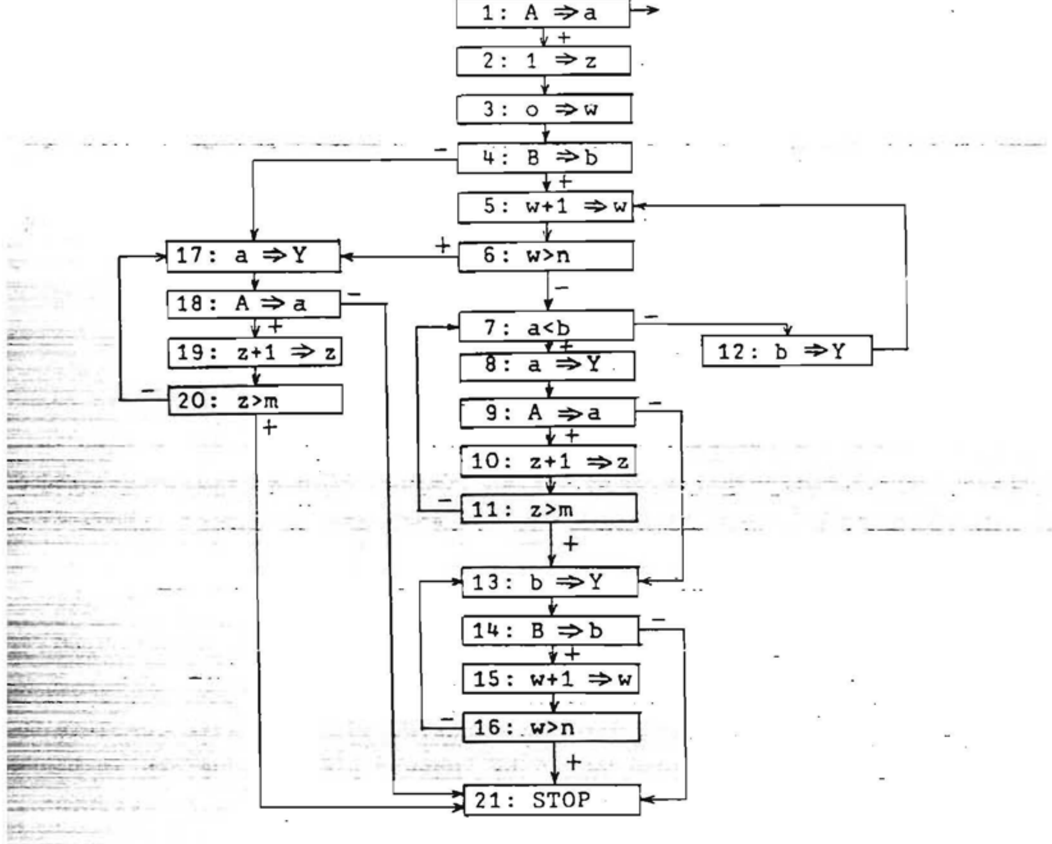


Fig. 1

By branch of program we understand a linear path between two conditional instructions (only the first instruction of path can be conditional). E.g., the program in Fig.1 has branches (7+,8),(6+,7) etc.

Let program have input files A,B,... and parameters m,n,... By a sample we shall understand fixed values of all these input variables: $A=A^{o}$, $B=B^{o}$,..., $m=m^{o}$, $n=n^{o}$,... .

Let us say that sample P realizes the branch h of program T if this branch is executed while running program T on sample P. E.g., when program in Fig.1 is run on the sample A=(1,2,3), B=(2,3), m=1, n=1, the path (1+,2,3,4+,5,6-,7+,8,9+,10,11+,13,14+,15,16+) containing branches (1+,2,3), (4+,5), (6-), (7+,8),... is executed.

Sample system is said to be complete for the given program, if every branch realizable by some arbitrary sample is realized by some sample in this system. Evidently, for every program there exists a finite complete sample system (CSS). The main problem is to find this system by means of some algorithm.

THEOREM 1. There is an algorithm constructing a finite complete sample system for every program in L_o.

In this case it is also decidable whether or not the program can loop infinitely on some sample (see [1] and [2]).

The following theorems show that Theorem 1 reveals in some sense the maximal boundaries for problem of constructing CSS to be solvable.

Let us consider a language L_1 in which counter values can also be compared with records of input files. Formally L_1 is obtained from L_o by adding new instructions of type $z<t$ and $t<z$, t being a main variable and z a counter.

THEOREM 2. There exists no algorithm constructing a finite complete sample system for every program in L_1.

Now let us consider a language L_2 where values of counters can be both increased by 1 and decreased by 1. Formally, L_2 is obtained from L_o by adding an instruction $z-1 \Rightarrow z$.

THEOREM 3. There exists no algorithm constructing a finite complete sample system for every program in L_2 (a subclass of programs in L_2 with two counters and no files and parameters is sufficient for non-existence)

Let us consider also a language L_3 in which a file can be reopened (i.e., input resumed from the first record). L_3 is obtained from the base language L_* by adding instructions of type REOPEN X.

THEOREM 4. There exists no algorithm constructing a finite complete sample system for every program in L_3 (a subclass of programs in L_3 with two input files with one usage of REOPEN for each of them is sufficient for non-existance).

HEURISTIC ALGORITHM.

Unsolvability in the abovementioned cases and complexity of algorithm used for proof of THEOREM 1 is caused by some artificial constructs improbable for real data processing programs. Now let us give a relatively simple heuristic algorithm not always yielding CSS for programs in L_o but at the same time applicable to more general language L.

In language L the counters are not specially singled out. Instead of this the following arithmetic instructions are applicable to all inner variables:

$t+u \Rightarrow v$
$t-u \Rightarrow v$
$t+c \Rightarrow v$
$t-c \Rightarrow v$
$c-t \Rightarrow v$

The other instructions of L are those of base language L_*.

Obviously L is a generalization of L_o, L_1, L_2, and construction of CSS is unsolvable for this language.

Now let us describe our heuristic algorithm. Let T be a program in L and $\alpha=(K_1,K_2,\ldots,K_r)$ a path in this program, K_i - an instruction with fixed exit (+ or -). E.g., $\alpha=(1{:}A \Rightarrow a+, 2{:}1 \Rightarrow z, 3{:}0 \Rightarrow w, 4{:}B \Rightarrow b-,$ $17{:}a \Rightarrow Y, 18{:}A \Rightarrow a+)$ (or $\alpha=(1+,2,3,4-,17,18+)$ if only labels of instructions are used).

Our aim is to find a sample realizing the path α. For this purpose the following system of inequalities and equalities $N(\alpha)$ is related with path α:

$$N(\alpha)=\begin{cases} M_o \\ M_1 \\ \ldots \\ M_r \end{cases},$$

where $M_o = \begin{cases} t_o=0 \\ u_o=0 \\ \ldots\ldots \end{cases}$, t,u,... - inner variables, and

M_i is the subsystem corresponding to instruction K_i. Let X be an input file and t,u,v inner variables. Let t_k, u_l, v_m be variables denoting the values of variables t,u,v after the execution of path $\alpha_{i-1}=(K_1,\ldots,K_{i-1})$ and X_s the last record of file X read during it (at the beginning corresponding variables are t_o, u_o, v_o). Let c be a ~~constant or an input parameter (in the last case it is a variable)~~ and Λ - the "blank" symbol, being less than any integer by definition. System M_i is defined in the following manner:

1) If $K_i=(w:X \Rightarrow u-)$ then

$M_i=\{X_{s+1} < \Lambda$

2) If $K_i=(w:X \Rightarrow u+)$ and if $M_1, M_2, \ldots, M_{i-1}$ do not contain inequality $X_j < \Lambda$ for any j(i.e.,no instruction of type $X \Rightarrow \ldots$-has been perofrmed) then

$$M_i = \begin{cases} X_{s+1} > \Lambda \\ X_{s+1} = u_{l+1} \end{cases}$$

In this case new variables u_{l+1} and X_{s+1} are introduced which have the same sense for instruction K_{i+1} as u_l and X_s for K_i.

If inequality $X_j < \Lambda$ has occured already, then

$$M_i = \begin{cases} X_{s+1} < \Lambda \\ X_{s+1} > \Lambda \end{cases}$$

i.e., a contradictory system is chosen.

3) If $K_i=(w:t \Rightarrow u)$ (or $K_i=(w:c \Rightarrow u)$ then

$M_i=\{t_k=u_{l+1}$ (or $M_i=\{c=u_{l+1}$).

New variable u_{l+1} is introduced in this case.

4) If $K_i=(w:t<u+)$ (or $K_i=(w:c<u+)$, or $K_i=(w:t<c+)$ then

$M_i=\{t_k+1 \le u_l$ (or $M_i=\{c+1 \le u_l$, or $M_i=\{t_u+1 \le c$).

5) If $K_i=(w:t<u-)$ (or $K_i=(w:c<u-)$, or $K_i=(w:t<c-)$) then

$M_i=\{t_k \ge u_l$ (or $M_i=\{c \ge u_l$, or $M_i=\{t_k \ge c$.

6) If $K_i=(w:t+u \Rightarrow v)$ (or $K_i=(w:t-u \Rightarrow v)$) then

$M_i=\{t_k+u_l=v_{m+1}$ (or $M_i=\{t_k-u_l=v_{m+1}$).

If $K_i=(w:t+c \Rightarrow v)$ (or $K_i=(w:t-c \Rightarrow v)$, or $K_i=(w:c-t \Rightarrow v)$)then

$M_i=\{t_k+c=v_{m+1}$ (or $M_i=\{t_k-c=v_{m+1}$, or $M_i=\{c-t_k=v_{m+1}$).

Equalities introduced by means of 6) we shall call arithmetic equalities.

It can be shown that there is an algorithm deciding the existence of an integer solution for system $N(\alpha)$ and constructing such a so-

lution in case of existence. For this purpose we can use, e.g., Gomory's algorithm for integer linear programming. Obviously, every solution gives a sample realizing the path α (it suffices to consider the values of input parameters and records of input files only).

Gomory's algorithm can also be used to show the existence of an algorithm for the following problem.

Let $\alpha=(K_1,\dots,K_r)$,

$$N(\alpha) = \begin{cases} M_o \\ \dots \\ M_{r-1} \\ x+1\le y \end{cases} \quad \text{(respectively, } N(\alpha) = \begin{cases} M_o \\ \dots \\ M_{r-1} \\ x\ge y \end{cases}\text{)}$$

and let system $\begin{cases} M_o \\ \dots \\ M_{r-1} \end{cases}$ have a solution, but

$$\begin{cases} M_o \\ \dots \\ M_{r-1} \\ x+1\le y \end{cases} \quad \text{(respectively } \begin{cases} M_o \\ \dots \\ M_{r-1} \\ x\ge y \end{cases}\text{)}$$

have no solution. Then the minimal p must be found such that the following system has a solution:

$$\begin{cases} M_o \\ \dots \\ M_{r-1} \\ x+1\le y+p \end{cases} \quad \text{(respectively } \begin{cases} M_o \\ \dots \\ M_{r-1} \\ x+p\ge y \end{cases}\text{)}$$

In this case we define this value of p to be the deficiency of instruction K_r on the path α.

Let us denote by $N'(\alpha)$ a system obtained from $N(\alpha)$ by deleting all arithmetic equalities. Now let us exclude variables w_1 from $N'(\alpha)$ such that in $N(\alpha)$ there is w_n with $n>1$. The exclusion is performed by adding some new inequalities and deleting all inequalities containing w_1. New inequalities are added in the following manner: If $N'(\alpha)$ contains $x+p_1\le w_1$ and $w_1+p_2\le y$ then $x+(p_1+p_2)\le y$ is added; if $x\ge w_1+p_1$ and $w_1\ge y+p_2$ then $x\ge y+(p_1+p_2)$ is added. Further, if $p_1+p_2>C_o=C_1-C_2$, where C_1 is the maximum and C_2 the minimum constants of the program, then p_1+p_2 is changed to C_o+1 in the new inequality. After the exclusion of all such w_1 we have a system where only one representative has remained from the group of variables differing only by subscripts (this representative has maximal subscript). Let us denote this system by $S(\alpha)$ and call it a state. Two states are said to be equal if they coincide after dropping subscripts of variables. Obvi-

ously, the number of different states is finite for every program.

The following assertion characterizing the meaning of the state can be proved:

Let α and β be paths such that:

a) α and β do not contain arithmetic instructions,

b) α and β end with the same instruction K,

c) $S(\alpha)=S(\beta)$.

Let γ be a path beginning at the instruction K.

Then path $\alpha+\gamma$ is realizable if path $\beta+\gamma$ is realizable. Moreover, $S(\alpha+\gamma)=S(\beta+\gamma)$.

Now let us define an ordering of paths. Path $\alpha=(K_1,K_2,K_3,\dots)$ is greater than path $\beta=(K'_1,K'_2,K'_3,\dots)$ if there is such i, that $K_1=K'_1$, $\dots,K_i=K'_i,K_{i+1}=(k+)$ and $K'_{i+1}=(k'-)$ where k and k' is the same instruction in the program.

Let us construct a special family of paths $(\alpha^1,\alpha^2,\dots,\alpha^p)$, beginning at the first instruction of program. Let paths $\alpha^1,\dots,\alpha^{j-1}$ be already constructed. Let us describe the construction of α^j. The least path α greater than α^{j-1} is considered (if j=1 we consider the absolutely least path α). More precisely, we construct this path α stepwise, adding new branches and testing on every step the stopping conditions described below. After the stopping we get the path α^j. We continue this procedure of path constructing until all paths are exhausted.

Now let us describe the stopping conditions. Let $\alpha=(K_1,K_2,K_3,\dots)$. To every instruction K_i we attach a state S_i equal to $S(\alpha_{i-1})$ where $\alpha_{i-1}=(K_1,\dots,K_{i-1})$.

<u>CONDITION 1</u>. Path α is stopped after the instruction K_i if system $N(\alpha_i),\alpha_i=(K_1,\dots,K_i)$, has no solutions, i.e. path α_i is not realizable. In this case we shall say that path $\alpha^j=\alpha_i$ is obtained by the condition C1.

In order to describe the other conditions we shall at first define the rules of writing and errasing $*$ at instructions in the program. Let conditional instruction K_i be equal (k_i+) (respectively (k_i-)). Then define $\overline{K}_i=(k_i-)$ (respectively (k_i+)).

(1) If $K_i=(k_i\,\varepsilon)$ is a comparison instruction, path $(K_1,\dots,K_{i-1},K_i)$ is realizable, path $(K_1,\dots,K_{i-1},\overline{K}_i)$ is not realizable, and instruction k_i has no $*$ at it in the program, then we write $*$ at k_i at this moment, i.e., at the moment when we have reached instruction K_i during our process of constructing the path α.

(2) $*$ is erased at k_i when we have reached an instruction $K_l=\overline{K}_i$ ($K_i=(k_i\,\varepsilon)$ must be the instruction in the path α which caused $*$ at

k_i) during the process of construction of path $\alpha=(K_1,K_2,\dots,K_l)$ and the path $(K_1,K_2,\dots,K_l)$ is realizable. (At this moment no new * is written according to (1)).

So the same instruction k_n in the program may have * at it at some moments and have not at other moments. If we consider a fixed moment, e.g., the one corresponding to some instruction K_i in path α then some instructions in program will have * at them and some will not. If at a given moment instructions k_n and k_m have * at them, then we shall say that instruction k_n is older than k_m if the current * at k_n has appeared earlier than that at k_m. So at some moment k_n may be older than k_m and at some other moment otherwise.

Let us consider the moment when instruction K_i on the path α is reached. Let some instructions in the program have * at them and therefore some age relation be defined among instructions.

CONDITION 2. Let S_i be the state attached to the instruction K_i. We move backwards along the path α and look for the first instruction K_m, $m<i$, such that $K_m=K_i$ and $S_m=S_i$. If we find such an instruction and also the following holds: no instruction $k_m,k_{m+1},\dots,k_i$ lying between k_m and k_i in the path α has * at it in the program, then path α is stopped after K_i.

CONDITION 3. Let $K_i=(k_i\ \ \varepsilon_i)$ be comparison instruction and let k_i have * at it. We move backwards along the path α and look for the preceding occurence of instruction k_i in the same state S_i, i.e., look for $K_m=(k_m\ \ \varepsilon_m)$ where $k_m=k_i$ and $S_m=S_i$. If $\varepsilon_m=\varepsilon_i$ holds (otherwise the rule (2) of erasing * is applicable and no stop occurs) then we find deficiencies p_m and p_i of instructions K_m and K_i. If $p_i\geq p_m$ then path is stopped after K_i.

CONDITION 4. Let S_i be the state attached to K_i. We move backwards along α and look for an instruction K_m, $m<i$, such that $K_m=K_i$ and $S_m=S_i$. If we find such an instruction and also the following holds: there are two instructions k_r and k_n in the program such that at the moment we consider K_i k_r is older than k_n (hence, they both have *) but at some previous moment k_n has been older than k_r, then α is stopped after K_i.

LEMMA. For every program in language L every path is stopped in finite number of steps according to Conditions 1-4.

Hence it follows that the family of paths $(\alpha^1,\alpha^2,\dots,\alpha^p)$ described before is finite indeed.

Now let us denote by $(\beta^1,\beta^2,\dots,\beta^s)$ the family of paths which is obtained from family $(\alpha^1,\alpha^2,\dots,\alpha^p)$ by deleting all paths α^j stopped

according to condition C1. Obviously all $\beta^1, \beta^2, \ldots, \beta^s$ are realizable. Solutions of corresponding systems of inequalities $N(\beta^1), \ldots, N(\beta^s)$ allow us to construct a sample system rel izing all these paths. This system is denoted by $\sum T$, T being a program. It follows from the Lemma that the algorithm described here converges in a finite number of steps for every program T in language L.

Now let us formulate an effectively testable completeness condition for our sample system $\sum T$.

We say that path α is realizable ignoring arithmetic if system $N'(\alpha)$ obtained from $N(\alpha)$ by deleting all arithmetic equalities has a solution. Let us denote by $(\gamma^1, \gamma^2, \ldots, \gamma^t)$ the family of paths which is obtained from $(\alpha^1, \ldots, \alpha^p)$ by scratching all paths not realizable ignoring arithmetics. Obviously $(\beta^1, \ldots \beta^s) \subseteq (\gamma^1, \ldots, \gamma^t)$. During the previous construction every occurence of instruction K in family $(\alpha^1, \ldots, \alpha^p)$ has a state S attached to it. In this case we shall say that pair (K,S) is contained in the family $(\alpha^1, \ldots, \alpha^p)$. Further, let us say that two pairs (K_i, S_i) and (K_j, S_j) ar equal if, firstly, $K_i = K_j$, i.e., K_i and K_j are the same instruction in the program with equally fixed exit, and, secondly, $S_i = S_j$.

THEOREM 5. If the set of different pairs (K,S) contained in the family $(\beta^1, \ldots, \beta^s)$ coincides with the set of different pairs (K,S) contained in the family $(\gamma^1, \ldots, \gamma^t)$ then the sample system $\sum T$ is complete for program T.

Real programs as a rule satisfy this completeness condition.

In particular, if the program contains no arithmetic instructions or contains the ones not affecting the values of variables used in conditional instructions then the above mentioned algorithm always yields complete sample system.

For the program of Fig.1 this algorithm constructs the following family of paths (only realizable paths containing new branches are given here, in order of their appearance):

β^1=(1+,2,3,4-,17,18-,21)
β^2=(1+,2,3,4-,17,18+,19,20-,17,18-,21)
β^4=(1+,2,3,4-,17,18+,19,20-,17,18+,19,20+,21)
β^6=(1+,2,3,4+,5,6-,7-,12,5,6-,7-,12)
β^7=(1+,2,3,4+,5,6-,7-,12,5,6+,17,18-,21)
β^{12}=(1+,2,3,4+,5,6-,7+,8,9-,13,14-,21)
β^{13}=(1+,2,3,4+,5,6-,7+,8,9-,13,14+,15,16-,13)
β^{14}=(1+,2,3,4+,5,6-,7+,8,9-,13,14+,15,16+,21)
β^{15}=(1+,2,3,4+,5,6-,7+,8,9+,10,11-,7-,12,5,6-,7-,12)

B^{26}=(1+,2,3,4+,5,6-,7+,8,9+,10,11+,13,14-,21)

The following sample system corresponds to it:

P_1:A=(1)	B=( )	m=1	n=1;
P_2:A=(1,2)	B=( )	m=2	n=1;
P_4:A=(1,2,3)	B=( )	m=2	n=1;
P_6:A=(2)	B=(1)	m=1	n=2;
P_7:A=(2)	B=(1)	m=1	n=1;
P_{12}:A=(1)	B=(2)	m=1	n=1;
P_{13}:A=(1)	B=(2,3)	m=1	n=2;
P_{14}:A=(1)	B=(2,3)	m=1	n=1;
P_{15}:A=(1,3)	B=(2)	m=2	n=2;
P_{26}:A=(1,3)	B=(2)	m=1	n=1;

It follows from the construction that conditions of Theorem 5 are satisfied and the constructed sample system is complete. It can be easily seen that this sample system reveals the bug in the program.

The described algorithm is far from optimum in general. It is considered here to illustrate a possible direction in studies related with practical implementation of complete sample system construction. It is plausible that an algorithm not too complex for implementation yielding CSS for practicallyall real programs can be constructed.

Let us note in conclusion that construction of CSS is an approach to application of incomplete induction principle in programming. An another usage of this principle is provided by synthesis of programs given by sample computations (see, e.g., [3] , [4] , [5]), though the questions of practical implementation are less clear there.

REFERENCES

1. Барздинь Я.М., Бичевский Я.Я., Калниньш А.А., Построение полной системы примеров для проверки корректности программ. Ученые записки Латвийского государственного университета,т.210 (1974), 152-187.
2. Калниньш А.А., Бичевский Я.Я., Барздинь Я.М., Разрешимые и неразрешимые случаи проблемы построения полной системы примеров. ——, 188-205.
3. Барздинь Я.М., Замечание о синтезе программ по историям их работы. -------, 145-151.
4. Barzdin J.M.,Synthesizing programs given by examples.-Lecture Notes in Computer Science 5 (A.Ershov,Ed.),pp.56-63,Springer-Verlag,Berlin,1974.
5. Bierman A.W., u.c., Automatic program synthesis.Technical Report, Ohio State University, 1973.

Translation of Russian references

1. Barzdin, J.M., Bičevskis, J.J. and Kalninsh, A.A., Construction of complete sample system for testing correctness of programs. Učenye zapiski Latv.gos. univ. 210 (1974), 152-187.

2. Barzdin, J.M., Bičevskis, J.J. and Kalnish, A.A., Solvable and unsolvable cases of the problem of construction of a complete sample system. Ibid, 188-205.

3. Barzdin, J.M., A note on synthesis of programs from their computational histories. Ibid, 145-151.

SDL '89
THE LANGUAGE AT WORK

Proceedings of the Fourth SDL Forum
Lisbon, Portugal, 9–13 October, 1989

edited by

Ove FÆRGEMAND
TFL
Hørsholm, Denmark

Maria Manuela MARQUES
INESC
Lisbon, Portugal

1989

NORTH-HOLLAND
AMSTERDAM • NEW YORK • OXFORD • TOKYO

SDL '89: The Language at Work
O. Færgemand and M.M. Marques (Editors)
© Elsevier Science Publishers B.V. (North-Holland), 1989

SDL TOOLS FOR RAPID PROTOTYPING AND TESTING

J. M. Barzdin, A. A. Kalnins, M. I. Auguston

Computing Center of the Latvian
State University
Blvd. Rainis 29
Riga 226250
USSR

The paper presents SDL tools which are being implemented at the Computing Center of the Latvian State University in cooperation with some other institutes. These tools are based on extended version of SDL'88 oriented towards executable specifications and include SDL compiler and other support tools for specification testing. The methodology of SDL use and SDL training is also discussed. In the conclusion problems of target code generation from SDL and other future plans are sketched.

1. INTRODUCTION.

The interest in the Soviet Union about the specification and design language SDL for the informal description and simulation of telecommunication systems has arisen long ago. This interest especially arose in the mid eighties in connection with the development of new generation of telecommunications systems. These systems appeared to be much more complex, and therefore the necessity arose for prototyping and testing their functions before the implementation of their software. A completely formal specification language for this purpose was necessary. SDL'88 meet these demands to a great extent. Therefore many institutions in the Soviet Union started rapid development of SDL tools. The paper describes SDL tools developed by the Computing Center of the Latvian State University (CC, LSU) in cooperation with some other institutes. At the present stage these tools seem to be the most advanced in the Soviet Union. They include graphical editor, SDL compiler and debugging and testing tools.

2. THE CHOICE OF THE SDL VERSION

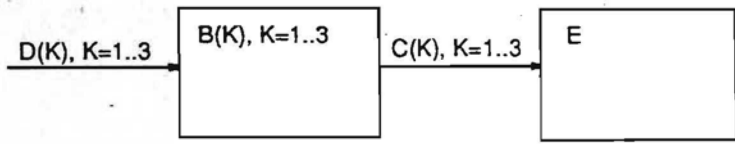
The first problem encountered during this project was the choice of SDL version. This SDL version should supply the user with convenient means for the design of formal specifications executable on computer. These executable specifications serve as a prototype for the system under development . The complete SDL'88 is too complex for executable implementation, especially its abstract data types. On the other side it does not contain many important facilities for the design of formal specifications. In this connection a modified version of SDL was developed (named SDL/PLUS [1]) to support the design of executable specifications. SDL/PLUS contains several extensions to SDL'88 and essential changes of its data handling part.

2.1. Extensions of SDL'88

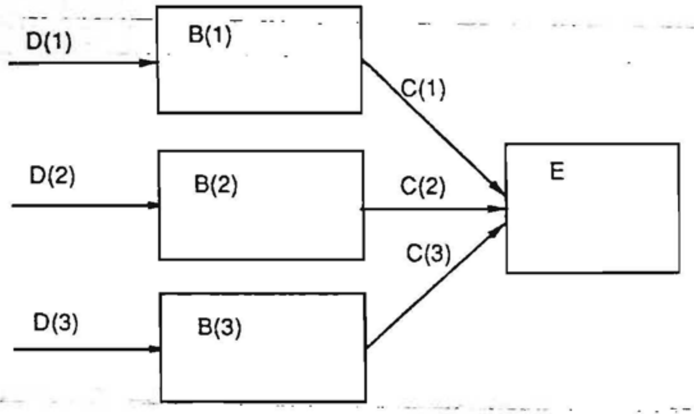
SDL'88 structuring facilities are perfect to describe the physical structure of the system (block and channel substructures etc.). Yet the process structuring facilities are unsufficient for the design of real systems. The introduction of service concept in SDL'88 improved the situation slightly but problems still remain.

During the top-down design of a complex system it is convenient to introduce large processes to be decomposed into components in the further stages of design. For this purpose the concept of subprocess is introduced in SDL/PLUS. It means that every process can be decomposed into smaller processes by the means of process substructure diagram. Signal routing means are also slightly extended for this purpose.

Real systems frequently contain many occurences of the same block or channel. For instance, digital switch may have many incoming trunks and each of the trunk may be served by its own microprocessor described as a block in SDL. SDL'88 has no means for the description of number of uniform blocks or channels. For this purpose the concept of block or channel array is introduced in SDL/PLUS. For example, system diagram in SDL/PLUS can contain the following fragment:



which is equivalent to the following:



SDL/PLUS allows also several input ports for the same process.

Some other important extensions to SDL'88 are supposed to write adequate specifications for implementation in a specific target environment. For this purpose the following concepts are introduced in SDL/PLUS:

- quasiparallel block, i. e. , a block corresponding to one CPU and therefore all its processes are executed in quasiparallel,

- process priorities, i. e. , processes in quasiparallel block are scheduled according their priorities,

- shared variables in quasiparallel blocks and their protection facilities,

- significant signals corresponding to interrupts in computer and causing instantaneous rescheduling of processes.

SDL/PLUS has also some less important extensions, e. g. , graphical loop symbol, not discussed here in detail.

2.2. Data Handling Means

Experience shows that abstract data types in SDL are not widely accepted by the users due to their complexity. Actually ADT are replaced by data handling part of the target language in all major executable SDL implementations [2]. The version of SDL/PLUS implemented at the present stage uses Pascal for data handling.

The situation on the spot is the following - the target language often is either of very low level or is chosen in the late stages of design. As large projects specially require completely formal and testable specifications the use of Pascal is approved by its wide usage as a data handling language.

Choice of some other high level target language(CHILL, C) also requires the usage of its data handling part in SDL. All our tools are designed so that the transition from Pascal to some other language requires just some months.

This is achieved by implementing all language processing components in a special high level compiler writing language RIGAL [3], also developed in the CC, LSU.

3. SDL OPERATING SYSTEM

SDL operating system has also been developed for use on host computer in simulated time mode. Several CPU's can be simulated - they must be described as several blocks in the system executed in parallel. Quasiparallel execution of processes in a block is assumed and priority scheduling of processes[1] is supported. Time simulation is implemented to support all timer operations. Time is advanced by a little increment at every signal sending/reception. There are also user controlled means for time advancing at other SDL statements. Every block has its own simulation time counter.

The aim of the SDL operating system is to guarantee completely correct externaly visible sequence of events according to SDL semantics defined more precisely in [1].The number of context switches from block to block was shown to be minimal for the selected scheduling algorithm. This is very important since executable SDL is intended also for system simulations with run time being critical.

No scheduling or timer services of the base operating system are used. At the moment the operating system accepts Pascal as the implementation language for SDL, but actually it is language and base operating system independent.

4. THE TOOL IMPLEMENTATION LANGUAGE RIGAL

The necessity of SDL implementation portability and need to get a working prototype in a short time implied the use of problem-oriented high level language for compiler development. As none of the languages familiar to us of this class met the requirements, a new language RIGAL was designed. It is a simple and powerful high level language for compiler writing. Data structures comprise atoms, lists and trees. Control structures are based on advanced pattern matching. Operations, such as table creation, code generation, message output, are executed simultaneously with parsing, like in YACC [4] and CDL-2 [5].

RIGAL ia a closed language where all the processing steps of compiling can be written without semantic subroutines in some other languages.

The language has means for easy usage of attribute grammar ideas and supports the style of recursive descent.The RIGAL has a special reference facility, that solves the global attribute problem better than ordinary attribute grammars.

The language provides tree manipulation facilities, including tree grammars.

All that makes possible to use RIGAL for syntactic analysis, program optimization, code generation and for preprocessor and convertor writing. RIGAL supports design of multi-pass compilers.

The interpreter of RIGAL was written in Pascal, and then an optimizing compiler RIGAL--> Pascal was written in RIGAL itself.

The optimizing RIGAL compiler in the VAX/VMS environment makes it possible to implement a production quality SDL--> Pascal compiler.

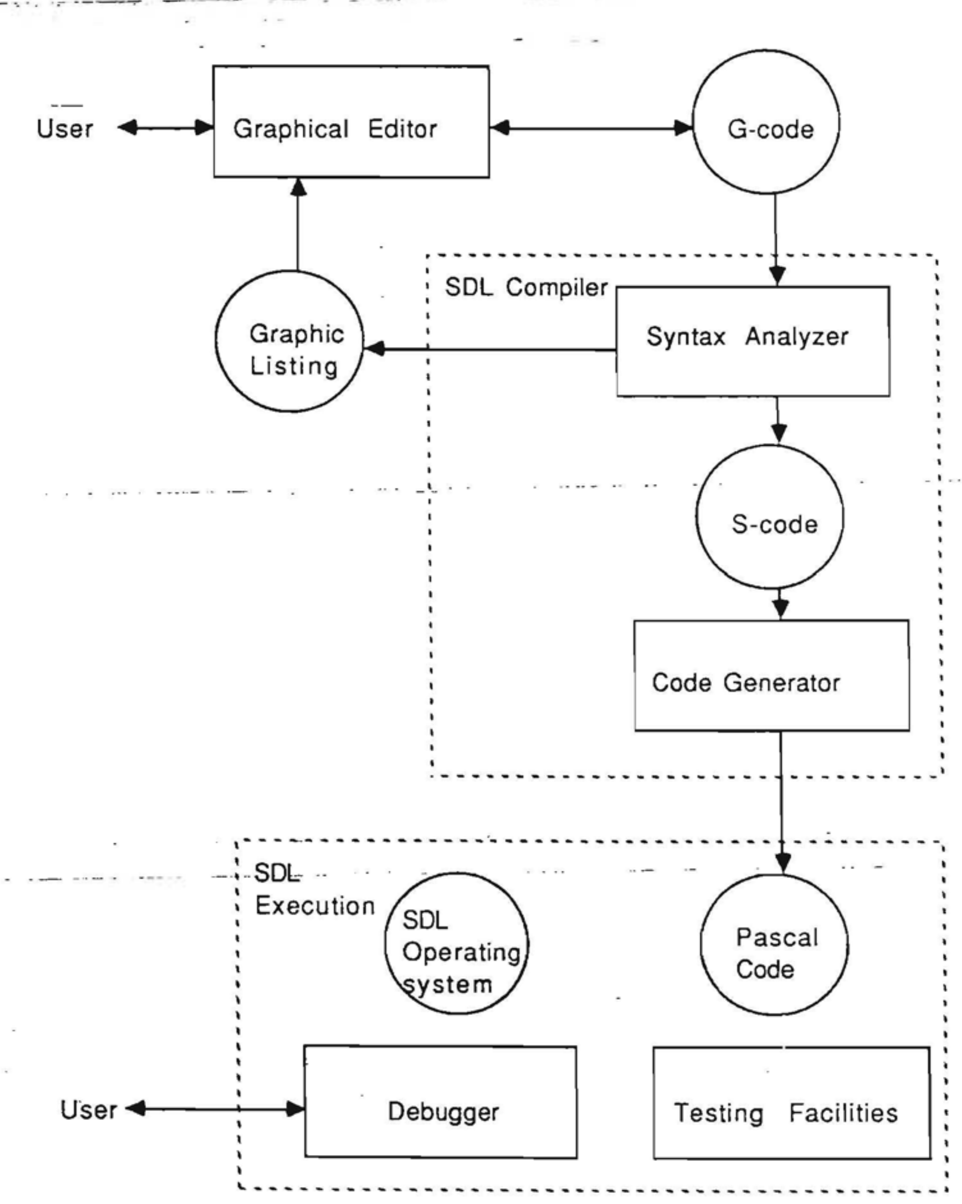


FIGURE 1. SDL support tools

5. SDL TOOLS AND THEIR IMPLEMENTATION

A set of SDL tools has been developed at CC, LSU to support the usage of SDL at various software design phases. Tools are implemented in the VAX/VMS environment. IBM PC's may be attached to VAX as graphic workstations. Figure 1 shows an overview of SDL tools implemented up-to-now.

5.1. Graphical Editor

Just the graphical form of SDL is used. The graphical SDL input uses IBM PC as a graphic workstation for VAX. The SDL graphical editor under MS DOS is also developed by CC, LSU but it has standard facilities common to many editors and will not be described in detail. Means to obtain hardcopies of SDL graphs are also available. SDL graphs are stored as disk files in a special compressed form called G - code.

5.2. SDL Compiler

The main tool in the set is SDL compiler for obtaining executable VAX code from SDL system (via Pascal).The SDL compiler is implemented in RIGAL. It consists of two parts.

5.2.1. Syntax Analyzer

The first part is the SDL syntax analyzer which includes also a transformer from G - code to internal PR form used as input for the analyzer. The syntax analyzer performs SDL syntax and static semantics checking including Pascal statements in SDL - type checking in assignments, signal inputs and outputs, signal and process name visibility etc. Error messages in process graphs are shown by means of graphical listing - a graph with error messages added at appropriate statements. The graphical editor has a special mode for easy observation of error messages.

The syntax analysis is performed in descending order - first for system, then for blocks, concluding with processes and procedures.

5.2.2. Intermediate code

The output of syntax analyzer is intermediate code called S - code in the form of abstract syntax tree with the necessary attributes added and coded as RIGAL data structure. S - code now is passed to code generator but it will be used also by other tools - static analyzer, target compiler, etc. S - code appears to be a very convenient form for storage of SDL objects and is easily processable by RIGAL statements. S - code for the system serves as a data base to maintain integrity during separate compilations of processes.

5.2.3. Code Generator

The second part of the compiler is the code generator producing standard VMS Pascal code from S - code. Code generator is applied to each of the processes.An external Pascal procedure is obtained from SDL process. The procedure body consists of CASE statement with a branch corresponding to each of the transitions. State/signal/transition table is held in a special coded form. This appears to be the optimal form for coding SDL in Pascal. All scheduling and timer operations as well as actual signal sending are compiled into calls to corresponding SDL operating system modules. The current version of SDL operating system assumes coroutine implementation of process instances thus the equivalents of Modula - 2 NEWPROCESS and TRANSFER procedures are introduced as runtime stack swappings (implemented in VAX MACRO).

Code generator automatically invokes VAX Pascal compiler, so an object module is produced for each of the processes, usually without Pascal diagnostics.

After that the code generation for the system itself is performed. Some Pascal procedures for system and blocks are generated and compiled. The Linker command line is generated and executed

to link all the produced modules together with SDL operating system modules. So the result of code generation for system is an executable VMS task.

At process corrections they can be recompiled separately and in most cases only system relinking is necessary.

At present stage Pascal is used as the host language for executable SDL because of use of Pascal data handling in SDL. In the case of another base language choice the code generator can be easily modified due to convenience of RIGAL both for tree processing and the description of code generation rules.

5.3. Testing and Debugging Facilities

Executable task obtained from SDL system is mainly intended for testing the SDL description of the system before its target implementation. Tools for setting up SDL testing environment are developed. They allow a convenient input from terminal or file of input signal sequences (together with their parameters). Time delays are also user controllable. Comprehensive ckeck of output signals is supported by explicit displaying or recording them. Some tools are specially adapted for testing subscriber interfaces of SPC exchanges when specified in SDL. Multiterminal prototyping tool for this kind of systems is also available where each terminal simulates a telephone set. All these tools look like standardized SDL processes to be added to an open system to close it. Tools to measure completeness of testing are under development.

SDL debugger is also implemented to obtain run-time information from the SDL viewpoint - active processes, states, signal output/input etc. The most advanced debugger is built as a post debugger where all the execution event trace is recorded in a compressed way and then examined by the debugger (possibly in both directions), giving the user illusion of real observing the execution. This approach appears to be more efficient (though has some deficiencies, too). On-line debugger with more limited capabilities is also available.

5.4. Portability of Tools

The use of RIGAL for all parts of compiler supports its high portability. The other smaller components of tools are implemented in Pascal and C. RIGAL is being ported into IBM PC and all the tools will be ported to IBM PC under MS DOS in 1990. Porting to UNIX™ environments also seems highly possible.

6. THE METHODOLOGY OF SDL USE

The tools support the main phases of switching software design. The editor and hardcopy facilities support the use of SDL for semiformal design and documentation of the switching system functions.

The main emphasis is made on completely formal design of software, where all the logics of action is expressed in executable SDL. The tools allow to create an executable prototype of the system which can be thoroughly validated by testing it on the computer. The same SDL description allows simulation of the system at various workloads.

At present only manual transition from the validated design to target implementation is possible. But future plans for a soon target compiler will allow nearly automatic transition to target implementation.

7. USAGE

The described SDL tools have been successfully applied to the design and testing of algorithms for an experimental PBX with additional features.The transport, network and data link levels of a specialized OSI - like protocol have also been tested by our tools.Tools are being applied to some formal description and testing of algorithms of some other telecommunications and switching

networks, e. g., a specialized local network. We suppose to use these tools for checking and testing some standard signalling systems and protocols, e. g., the most important parts of signalling system No.7. Hence the description of this signalling (CCITT recommendations Q701-741) is a somewhat informal in its data handling part, some formalization of this part is to be done before testing. This will allow to test the description exhaustively on computer.

8. SDL TRAINING

A textbook on SDL (in Russian) is prepared in the CC, LSU. Experience shows SDL to be a relatively complex language to learn - even without ADT. Therefore the layout of SDL in the textbook is divided into levels according to user categories:

level 1 - local (stand-alone) process for informal use by algorithm designers,

level 2 - system structure for use by system designers,

level 3 - complete SDL'88, for use by program designers,

level 4 - extensions to SDL'88 for use by program implementors to describe implementation specific details.

9. FUTURE PLANS

As it was mentioned above tools can be easily adapted to some other target language - not only Pascal. The principles of SDL compiler generating code for some target environments are developed. The necessary optimization level will be provided using alternative code patterns for the same SDL construct. The optimization will be user-controlled by means of formal comments - pragmas at SDL declarations. Such a controllable compiler for a target environment will be available in 1990. Its development strongly relies on the possibilities of RIGAL. Actually only the code generation part has to be rewritten, besides many of the code generation algorithms of present compiler will also be retained.

The development of testing tools is also planned including SDL static analysis and automatic test case generation. We suppose to generate test case sets exercising all transitions in processes using methods similar to [6].

REFERENCES

[1] Barzdin J., Kalnins A., Strods J. and Sicko V., Specification Language SDL/PLUS and its Applications (Latvian State University, Riga, 1988, in Russian)

[2] Saracco R. and Tilanus P. A. J. (eds.), SDL'88. State of the Art and Future Trends (North-Holland, Amsterdam, 1987)

[3] Auguston M., The RIGAL Programming Language, "Programmirovanije", 1989, 4, (in Russian)

[4] Johnson S.C., YACC - yet Another Compiler Compiler. (Bell laboratories, Murray Hill, N. J., 1978, a technical manual)

[5] Koster C. H. A., Using the CDL Compiler Compiler. Lecture notes in Computer Science, 1977, Vol. 21.

[6] Barzdin J. M., Bicevskis J. M. and Kalninsh A. A., Automatic Construction of Complete Sample System for Program Testing, in: Proc. IFIP Congress 1977, (North-Holland, Amsterdam, 1977) pp. 57-62.

Institutt for Datateknikk og Telematikk
Division of Computer Systems and Telematics

Gruppe for Systemutviklingsteknologi
Systems Development Technology Group

Proceedings of the

Nordic Workshop

on

Programming Environment Research

Trondheim, June 11-12, 1990

Editors: Ole Solberg, Arne Venstad

ELAB-RUNIT
N-7034 Trondheim
Norway
Phone: +47 7 59 26 00
Fax: +47 7 53 25 86
Telex: 52 03 75 ELAB

IDT,NTH
N-7034 Trondheim
Norway
Phone: +47 7 59 34 40
Fax: +47 7 59 44 66

SDL SUPPORT ENVIRONMENT FOR PROTOTYPING AND TESTING

A.A.Kalnins

Computing Center of the Latvia
University
Blvd. Rainis 29
Riga 226250
USSR

The paper presents SDL support environment which is being implemented at the Computing Center of the Latvia University in cooperation with some other institutes. The environment comprises a set of tools which are based on extended version of SDL'88 oriented towards executable specifications and include SDL compiler and other support tools for specification testing. The methodology of SDL use and SDL training is also discussed. In the conclusion problems of target code generation from SDL and other future plans are sketched. The paper is a further development of [1].

1. INTRODUCTION

The interest in the Soviet Union about the specification and design language SDL for the informal description and simulation of telecommunication systems has arisen long ago. This interest especially arose in the mid eighties in connection with the development of new generation of telecommunications systems. These systems appeared to be much more complex, and therefore the necessity arose for prototyping and testing their functions before the implementation of their software. A completely formal specification language for this purpose was necessary. SDL'88 meet these demands to a great extent. Therefore many institutions in the Soviet Union started

rapid development of SDL tools. The paper describes SDL tools developed by the Computing Center of the Latvia University (CC.LU) in cooperation with some other institutes, mainly the Kvazar research association in Moscow. At the present stage these tools seem to be the most advanced in the Soviet Union. They include graphical editor, SDL compiler and debugging and testing tools.

2. THE PROPOSED SOFTWARE DEVELOPMENT PARADIGM

The main approach of this project is a consistent use of SDL in all phases of the software life cycle. It includes semiformal use of SDL in the early stages of system design supported only by tools for editing and documenting. Then follows the formal specification of system functions, which should be accompanied by thorough tool-based testing and prototyping of these functions. The main implementation dependent features should also be specifiable and testable at the SDL level. The transition to target implementation should be through automatic compilation including user guided optimizing to adapt the code to target environment peculiarities.

3. THE CHOICE OF THE SDL VERSION

The first problem encountered during this project was the choice of SDL version. This SDL version should supply the user with convenient means for the design of formal specifications executable on computer. These executable specifications serve as a prototype for the system under development. The complete SDL'88 is too complex for executable implementation, especially its abstract data types. On the other side it does not contain many important facilities for the design of formal specifications. In this connection a modified version of SDL was developed (named SDL/PLUS [2]) to support the design of executable

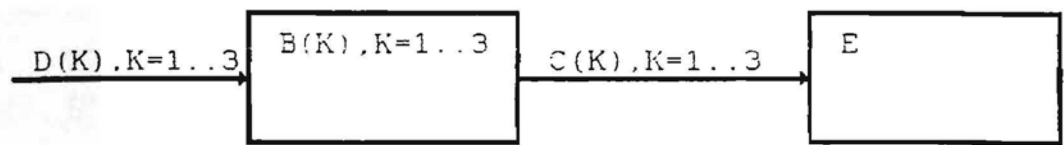
specifications. SDL/PLUS contains several extensions to SDL'88 and essential changes of its data handling part.

3.1. Extensions of SDL'88

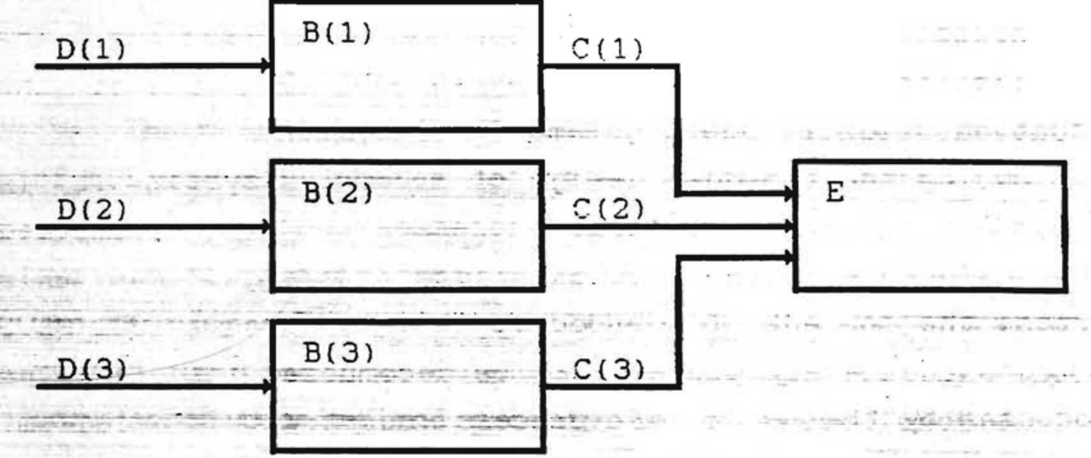
SDL'88 structuring facilities are perfect to describe the physical structure of the system (block and channel substructures etc.). Yet the process structuring facilities are insufficient for the design of real systems. The introduction of service concept in SDL'88 improved the situation slightly but problems still remain.

During the top-down design of a complex system it is convenient to introduce large processes to be decomposed into components in the further stages of design. For this purpose the concept of subprocess is introduced in SDL/PLUS. It means that every process can be decomposed into smaller processes by the means of process substructure diagram, arbitrary depth of decomposition is allowed. Signal routing means are also slightly extended for this purpose.

Real systems frequently contain many occurrences of the same block or channel. For instance, digital switch may have many incoming trunks and each of the trunk may be served by its own microprocessor described as a block in SDL. SDL'88 has no means for the description of number of uniform blocks or channels. For this purpose the concept of block or channel array is introduced in SDL/PLUS. For example, system diagram in SDL/PLUS can contain the following fragment:



which is equivalent to the following:



SDL/PLUS allows also several input ports for the same process. thus supporting, e.g., the description of a handler for various types of interrupts as one process.

Some other important extensions to SDL'88 are supposed to write adequate specifications for implementation in a specific target environment. For this purpose the following concepts are introduced in SDL/PLUS:

- quasiparallel lock. i.e., a block corresponding to one CPU and therefore all its processes are executed in quasiparallel.

- process priorities, i.e., processes in quasiparallel block are scheduled according their priorities.

- shared variables in quasiparallel blocks and their protection facilities,

- significant signals corresponding to interrupts in computer and causing instantaneous rescheduling of processes.

SDL/PLUS has also some less important extensions, e.g.. graphical loop symbol. not discussed here in detail. Some rarely used facilities in SDL'88. e.g.. export/import, are omitted on SDL/PLUS.

Part of the extensions to SDL'88 included in SDL/PLUS are mainly in line with the object oriented extensions to SDL proposed by Nordic countries.

3.2. Data Handling Means

Experience shows that abstract data types in SDL are not widely accepted by the users due to their complexity. Actually ADT are replaced by data handling part of the target language in executable SDL implementations [3]. Though SDL predefined data types are used in some implementations [4],the introduction of new data types is actually performed by means of user written procedures and functions in target language corresponding to abstract operation. So user actually describes data handling in the target language [C, Pascal, CHILL].

The current version of SDL/PLUS uses Pascal for data handling. This is due to the fact that in our cases the target language often is either of very low level or is chose in the late stages of design. As large projects specially require completely formal and testable specifications the use of Pascal is approved by its wide usage as a data handling language and so facilitates the testing of functional specifications with lesser efforts.

Choice of some other high level target language (CHILL, C) also requires the usage of its data handling part in SDL. All our tools are designed so that the transition from Pascal to some other language requires just some months.

This is achieved by implementing all language processing components in a special high level compiler writing language RIGAL [5], also developed in the CC.LU. A special care is also taken to isolate the data handling part of the language in both syntactic analysis and code generation.

4. SDL SUPPORT ENVIRONMENT AND ITS IMPLEMENTATION

SDL support environment called RIGA-SDL has been developed at CC. LU to support the usage of SDL at various design phases. The support environment consists of an

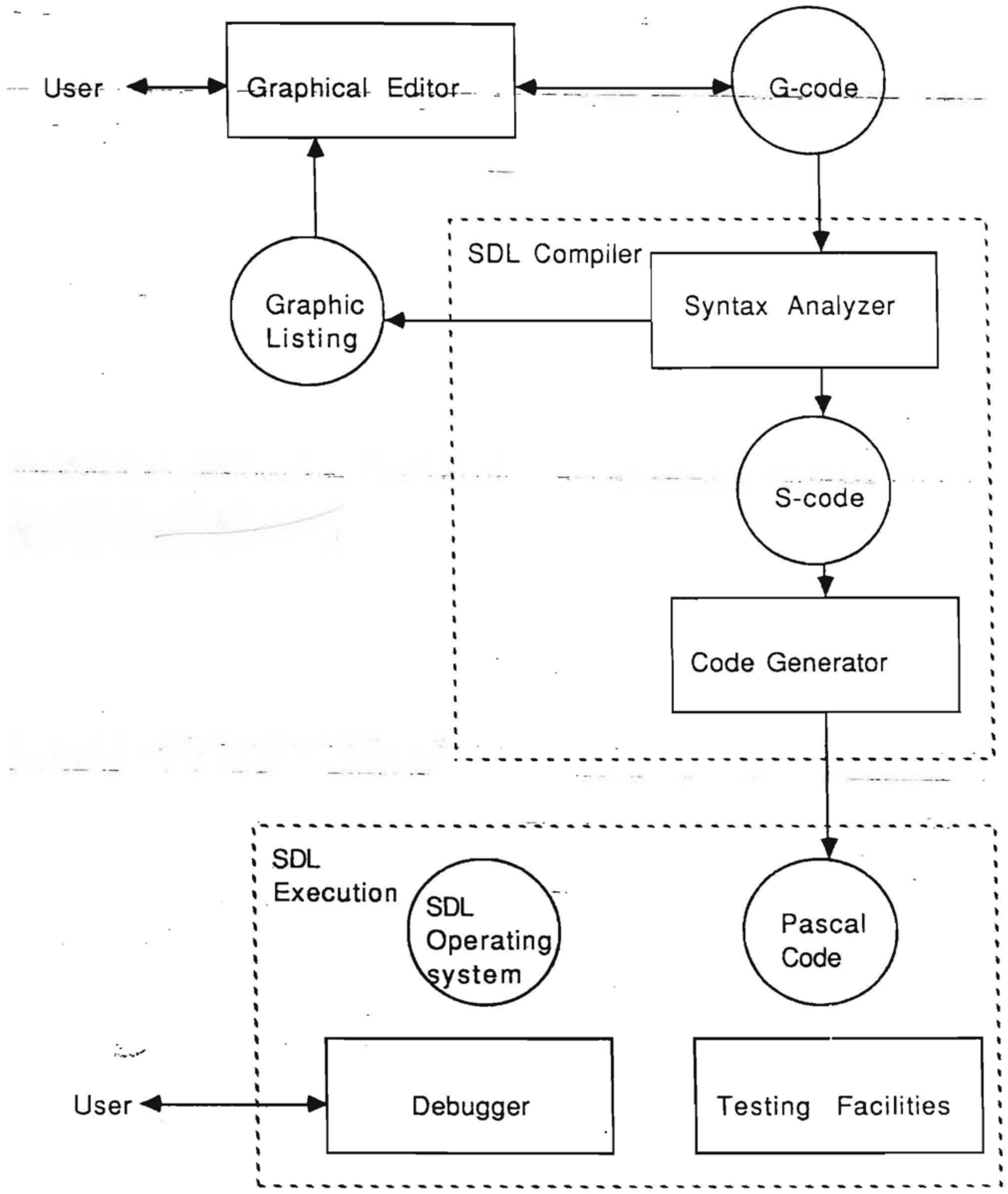


FIGURE 1. SDL support tools

integrated set of tools.

Tools are implemented in both VAX/VMS and IBM PC / MS DOS environments. A mixed environment where PC's are attached to VAX as graphic workstations is also possible. Figure 1 shows an overview of SDL tools implemented up to now.

4.1. User Interface

All tools on IBM PC are accessible from an integrated menu-driven interface. This interface gives an overview of SDL system under development in a tree-like form. It serves also as a simple project data base. The operations available for the selected SDL object (editing, compiling etc.) are shown by menus and submenus.

On VAX most of the operations are available from a single SDL command line naming the object to be processed.

4.2. Graphical Editor

Just the graphical form of SDL is used for process diagrams. The SDL graphical editor under MS DOS is developed by CC.LU. It has standard facilities common to many SDL editors and will not be described in detail. Means to obtain hardcopies of SDL graphs are also available. SDL graphs are stored as disk files in a special compressed form called G-code.

Graphical editor for block interaction diagrams is under development, for the moment this kind of information is entered via PR-form. There is also a simple graphical editor for VAX alphanumeric terminals.and PC with its editors can be used as graphic workstation for VAX.

4.3. SDL Compiler

The main tool in the set is SDL compiler for obtaining

executable code from SDL system (via Pascal). The SDL compiler is implemented in RIGAL. The compilation is performed in descending order - first for system, then for blocks, concluding with processes and procedures. SDL-compiler consists of two parts, acting as a two-pass compiler.

4.3.1. Syntax Analyzer

The first part is the SDL syntax analyzer which includes also a transformer from G-code to internal PR form used as input for the analyzer. The syntax analyzer performs SDL syntax and static semantics checking (including Pascal statements in SDL) - type checking in assignments, signal inputs and outputs, signal and process name visibility etc. Error messages in process graphs are shown by means of graphical listing - a graph with error messages added at appropriate statements. The graphical editor has a special mode for easy observation of error messages.

4.3.2. Intermediate code

The output of syntax analyzer is intermediate code called S-code in the form of abstract syntax tree with the necessary attributes added and coded as RIGAL data structure. S-code now is passed to code generator but it will be used also by other tools - static analyzer, target compiler, etc. S-code appears to be a very convenient form for storage of SDL objects and is easily processable by RIGAL statements. S-code for the system serves as a data base to maintain integrity during separate compilations of processes.

4.3.3. Code Generator

The second part of the compiler is the code generator

producing standard Pascal code from S-code.

It generates standard VMS Pascal for VAX and Turbo Pascal for IBM PC. An external Pascal procedure (unit) is generated for an SDL object (system. block. process). In the case of SDL process the procedure body consists of CASE statement with a branch corresponding to each of the transitions. State/signal/transition table is held in a special coded form. This appears to be the optimal form for coding SDL in Pascal. All scheduling and timer operations as well as actual signal sending are compiled into calls to corresponding SDL operating system modules. The current version of SDL operating system assumes coroutine implementation of process instances thus the equivalents of Modula-2 NEWPROCESS and TRANSFER procedures are introduced as runtime stack swappings.

Code generator automatically invokes Pascal compiler, so an object module is produced for each of the SDL objects, usually without Pascal diagnostics.

When user selects a Link operation, the Linker command line is generated and executed to link all the produced modules together with SDL operating system modules (or to assemble the compiled units in the case of Turbo Pascal). So the result of linking is an executable task functioning as an executable model for the SDL system.

At process corrections they can be recompiled separately and only system relinking is necessary after that. If a higher level object, e.g., block is changed, UNIX like MAKE-mode automatically ensures all the necessary recompilations.

At present stage Pascal is used as the host language for executable SDL because of use of Pascal data handling in SDL.In the case of another base language choice the code generator can be easily modified due to convenience of RIGAL both for tree processing and the description of code generation rules.

4.4. SDL Operating System

In order to support the execution of Pascal code generated by compiler,. SDL operating system has also been developed for use on host computer in simulated time mode.Several CPU's can be simulated - they must be described as several blocks in the system executed in parallel. Quasiparallel execution of processes in a block is assumed and priority scheduling of processes [2] is supported. Time simulation is implemented to support all timer operations. Time is advanced by a little increment at every signal sending/reception. There are also user controlled means for time advancing (DELAY statements). Every block has its own simulation time counter.

The aim of the SDL operating system is to guarantee completely correct externally visible sequence of events according to SDL semantics defined more precisely in [2]. The number of context switches from block to block was shown to be minimal for the selected scheduling algorithm. This is very important since executable SDL is intended also for system simulations with run time being critical.

No scheduling or timer services of the base operating system (VMS or MS DOS) are used. At the moment the operating system accepts Pascal as the implementation language for SDL, but actually it is language and base operating system independent.

4.5. Testing and Debugging Facilities

Executable task obtained from SDL system is mainly intended for testing the SDL description of the system before its target implementation. Tools for setting up SDL testing environment are developed. They allow a convenient input from terminal or file of input signal sequences (together with their parameters). Time delays are also user controllable. The storing of input signal sequences in a

file ensures the creation of comprehensive test sets for regression testing after system changes. Comprehensive check of output signals is supported by explicit displaying or recording them. The recording of both input and output signals at selected interfaces gives a highly observable representation of the system functioning in a form similar to sequence charts. Some tools are specially adapted for testing subscriber interfaces of SPC exchanges when specified in SDL. Multiterminal prototyping tool for this kind of systems is also available where each terminal simulates a telephone set. All these tools look like standardized generic SDL processes to be added to an open system to close it. Tools to measure completeness of testing or to generate stubs for processes not designed yet are under development.

SDL debugger is also implemented to obtain run-time information from the SDL viewpoint- active processes, states, signal output/input etc. The most advanced debugger is built as a post debugger where all the execution event trace is recorded in a compressed way and then examined by the debugger (possibly in both directions), giving the user illusion of real observing the execution. This approach appears to be more efficient (though has some deficiencies, too). On-line debugger with more limited capabilities is also available. Interfaces to standard Pascal debuggers are also provided for debugging of data handling parts.

4.6. Target compiler

In accordance with our software development paradigm an automatic transition from the validated design specification in SDL to target implementation is quite needed. The main problems encountered here are very specific optimization requirements and variety of target environments. A target compiler generating Pascal code for a real-time environment from the intermediate S-code is under development. The main facility adopted to improve the

optimization is pragmas. They are formal comments attached by designer to SDL declarations and statements to guide the code generation. Pragmas are used in cases where there are alternative code patterns for the same SDL construct whose selection is imposed by external requirements and in cases where designer supplied interface procedures should be preferred to the compiler generated ones. Prototype target compiler for extended MS DOS environment as a target will be completed in 1990. It will be used to produce software for an experimental network of PC's. Target compilers for other environments and languages are also considered.

4.7. Portability of Tools

The use of RIGAL for all parts of compiler supports its high portability. The other smaller components of tools are implemented in Pascal and C. Complete compatibility of RIGAL environments on VAX and IBM PC has provided a relatively easy porting of SDL compiler from VAX to PC disregarding the differences of Pascal versions. Porting the Rigal itself was also easy due to use of standard Pascal only. Porting to UNIX™ environments also seems highly possible.

5. THE METHODOLOGY OF SDL USE

An SDL based methodology for telecommunications software design has been developed [2]. It is tool based in all development phases. Some additional telephone switching oriented notations are proposed to facilitate the semiformal use of SDL in the early stages of design. The editor and hardcopy facilities support the use of SDL for semiformal design and documentation of the switching system functions.

The main emphasis is made on completely formal design of software, where all the logics of action is expressed in executable SDL. The tools allow to create an

executable prototype of the system which can be thoroughly validated by testing it on the computer. The same SDL description allows simulation of the system at various workloads. The major implementation details should be added to the validated functional specification. this possibility is supported by the abovementioned language extensions in SDL. Some additional testing of the implementation specification by the tools is desirable. An automatic transition to target implementation with minimum debugging at the target level should follow. In the absence of target compiler some methods of manual transition have been developed.

6. USAGE

The described SDL tools have been successfully applied to the design and testing of the subscriber interface part for an experimental PBX with additional features in accordance with CEPT standards. Most of the application has been performed by Kvazar research association. The transport, network and data link levels of a specialized OSI-like protocol have also been tested by our tools. Tools are being applied to some formal description and testing of algorithms of some other telecommunications and switching networks, e.g., an experimental PC based local network. We hope to produce the software for the latter system completely by means of the SDL support environment. Some applications in the area of ISDN and common channel signaling are also planned.

7. SDL TRAINING

A textbook on SDL (in Russian) is prepared in the CC,LU. Experience shows SDL to be a relatively complex language to learn - even without ADT. Therefore the layout of SDL in the textbook is divided into levels according to user categories:

level-1 local (stand-alone) process for informal use by algorithm designers.

level-2 system structure for use by system designers.

level-3 complete SDL'88, for use by program designers.

level-4 extensions to SDL'88 for use by program implementors to describe implementation specific details.

8. FUTURE PLANS

The main research perspectives are connected with automatic target code generation and improved testing facilities. The compiler perspectives include various target languages and more powerful optimization by means of flow analysis, more powerful pragma language and SDL level program transformations.

The main problem in the testing area is automatic test case generation from SDL specifications. We suppose to generate test case sets exercising all transitions in processes using methods similar to [6], thus taking into account also the data dependent conditions. We hope that these methods will allow more thorough testing than methods based solely on finite state machine approach [7], especially in protocol area. More advanced SDL static analysis tools are also considered.

REFERENCES

[1] Barzdin J., Kalnins A., Auguston M., SDL Tools for Rapid prototyping and testing, in Fourth SDL Forum, Lisbon Oct. 1989 (North-Holland, 1989) pp. 127 - 133.

[2] Barzdin J., Kalnins A., Strods J. and Sicko V., Specification Language SDL/PLUS and its Applications (Latvia State University, Riga, 1988, in Russian)

[3] Saracco R. and Tilanus P.A.J. (eds.), SDL'88. State of the Art and Future Trends (North-Holland, Amsterdam, 1987)

[4] Encontre V. GEODE: An Industrial Environment for Designing Real Time Distributed Systems in SDL. in Fourth SDL Forum. Lisbon Oct. 1989 (North-Holland, 1989) pp. 105 - 116.

[5] Auguston M.The RIGAL Programming Language. "Programmirovanije", 1989, 4, (in Russian)

[6] Barzdin J.M., Bicevskis J.J. and Kalninsh A.A. Automatic Construction of Complete Sample System for Program Testing, in: Proc. IFIP Congress 1977, (North-Holland, Amsterdam, 1977) pp. 57 - 62.

[7] Brömstrup L., Hogrefe D. TESDL: Experience with Generating Test Cases from SDL Specifications, in Fourth SDL Forum , Lisbon, Oct. 1989 (North-Holland, 1989) pp. 267 - 279.

J. Bārzdiņš D. Bjørner (Eds.)

Baltic Computer Science

Selected Papers

Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest

AUTOMATIC CONSTRUCTION OF TEST SETS: THEORETICAL APPROACH

Andrejs Auziņš, Jānis Bārzdiņš, Jānis Bičevskis, Kārlis Čerāns, Audris Kalniņš
The University of Latvia
Raiņa bulv. 19, Rīga 226250, Latvia

Abstract

We consider the problem of automatic construction of complete test set (CTS) from program text. The completeness criterion adopted is C_1, i.e., it is necessary to execute all feasible branches of program at least once on the tests of CTS. A simple programming language is introduced with the property that the values used in conditional statements are not arithmetically deformed. For this language the CTS problem is proved to be algorithmically solvable and CTS construction algorithm is obtained. Some generalizations of this language containing counters, stacks or arrays are considered where the CTS problem remains solvable. In conclusion the applications of the obtained results to CTS construction for real time systems are considered.

1 Introduction

Program testing remains the least automated and most resource–demanding step in the program development process. There are several testing methods: functional testing, structural testing, random testing, etc. In this paper we consider only structural testing. In the structural testing all activities, including test case selection, are based on program structure. The question about the completeness of the selected test set appears naturally. In the case of structural testing the most widely accepted completeness criterion is C_1 [11]: a test set is said to satisfy criterion C_1 if all feasible branches of program can be executed on this set. We shall not discuss how complete criterion C_1 is (see, e.g. [1,15]), we just note once more that this criterion is widely accepted in practice and there seems to be found no better criterion up to the moment.
For a fixed completeness criterion the problem of automatic construction of complete (with respect to the criterion) test set from program text arises. In this paper we consider the automatic construction of complete test sets according to criterion C_1. Such test sets will be simply called complete test sets (CTS), and the construction problem of such test sets will be called CTS problem.

We note just now that CTS problem is algorithmically unsolvable in general case, besides, as further results show, the algorithmic unsolvability appears swiftly. The aim of the paper is to find sufficiently large program classes with algorithmically solvable CTS problem and to develop the corresponding algorithms.
Yet, another remark. The variable value ranges are limited for real programming languages. For example, integer variable in Pascal can assume values from −2147483648 to 2147483647. These value limits formally yield the algorithmic solvability of CTS problem: the set of theoretically possible values of all internal variables of program can be used as the set of program states (this set will always be finite for the assumed restrictions), hence, CTS can be constructed by means of exhaustive search. However, it is clear that such a method is unusable in practice. A question arises how to exclude the trivial solution by means of exhaustive search. One of the ways is to drop the restrictions on variable value ranges. In this case the variable value range is infinite and thus the trivial solution to CTS problem by means of exhaustive search is excluded. If we, nevertheless, find an algorithm for CTS

construction, it is probable that this algorithm will not use exhaustive search. Therefore we can hope that this algorithm will not use exhaustive search also for finite value ranges. Namely this way will be used in the paper. The obtained results confirm that the CTS construction algorithms obtained this way don't use exhaustive search indeed and are practically usable in many cases.

To conclude the introduction we give brief characteristics of program classes for which the solvability of CTS problem has been proved and corresponding algorithms obtained. Firstly these program classes have the property that variable values used in conditional statements are not arithmetically deformed, i.e., these values are read directly from program input data. The second characteristic property of these classes is connected with some restrictions on direct access to data. An important class of programs is formed by programs with counters. The CTS construction problem is obviously unsolvable for programs with free use of counters. Nevertheless, sufficiently general program classes with counters having solvable CTS problem can be found. One of the most important of such classes with solvable CTS problem is programs with real time counter.

In the conclusion some methods are presented for reducing real time programs to the models considered.

The paper contains results obtained by the authors at various times [2–10], as well as new results. New results are presented in Section 5 (J.Bārzdiņš) and Sections 9, 10 (K.Čerāns).

2 The First Solvable Case: Programs in Base Language L_0

2.1 Description of Language L_0

In order to expose the principal ideas we introduce a very simple programming language L_0 for the processing of sequential files. Nevertheless, a large part of business data processing in the sequential files area can be formalized in this language (adequately enough to investigate the construction of complete test set). This language can be characterized by the fact that values taking part in comparison statements are undeformed (i.e., such as read from input). This restriction is acceptable in practice because it is typical for data processing programs that program logic is controlled only by input data (e.g., record type) and that these data are used in comparison statements undeformed.

Now let us describe the language L_0. Programs in L_0 use external variables of special type, named tapes. We shall use tapes to represent finite sequences of integers. We shall say that tape X contains a sequence of integers $(x_1, x_2, \ldots, x_r)$, if the first cell of the tape contains x_1, the second - x_2, . . . , the r-th - x_r, but the other cells are empty (fig.1).

X: | x_1 | x_2 | . . . . | x_r |

Fig. 1

To put it otherwise it means that the value of the variable X is $(x_1, x_2, \ldots, x_r)$ in this case. We shall denote the i-th cell of X by X_i, this notation being used also as an

integer - valued variable (the value of X_i is undefined if X_i is empty).

A program in $\mathbf{L_0}$ has a finite number of input tapes and a finite number of output tapes associated with it. The program processes the values of its input tapes into values of its output tapes.

Initially the reading (writing) head is located on the first cell. The execution of an input (output) statement moves the head one position right. A program also has a finite number of integer-valued internal variables. We assume that all internal variables are initialized to 0 in the beginning. Now let us describe the statements of $\mathbf{L_0}$. Let X be an input tape, Y - output tape, t, u - internal variables and c - constant (fixed integer). The following statements are available:

1. $\mathbf{X \rightarrow t}$. The current cell of tape X is assigned to variable t. Thus, if $X = (x_1, x_2, \ldots, x_p)$, the first occurrence of statement $\mathbf{X \rightarrow t}$ assigns the value x_1 to t, the second - x_2 and so on. The statement has two exits: "+" if the current cell is nonempty and exit "-" if the cell is empty (tape is exhausted). In the last case the value of t is not changed. (Input statement).

2. $\mathbf{t \rightarrow Y}$. The value of variable t is assigned to the current cell of tape Y. (Output statement).

3. $\mathbf{u \rightarrow t}$ (respectively $\mathbf{c \rightarrow t}$). The value of variable u (constant c) is assigned to variable t. (Assignment statement).

4. $\mathbf{u<t}$ (respectively $\mathbf{c<t}$, $\mathbf{u<c}$). The statement has two exits: if the value of u (respectively c) is less than the value of t (respectively c), then the exit "+" is used, otherwise, the exit "-". (Comparison statement).

5. **NOP**. Dummy statement (nothing is done). It is used instead of statements not essential for the construction of complete test set when more general programing languages are reduced to $\mathbf{L_0}$. (Informally, these are unconditional statements not affecting the variable values used in comparisons).

6. **STOP**.

Statements 1 and 4 having two exits are called conditional statements, the other ones are called unconditional.

Informally a program in $\mathbf{L_0}$ is a program constructed from the abovementioned statements in a normal way. Formally we define a *program in the language* $\mathbf{L_0}$ as a quadruple

$$(\mathbf{X}, \mathbf{Y}, \mathbf{Z}, \mathbf{P}),$$

where $\mathbf{X}$ is a set of input tapes (e.g., $\mathbf{X}$={ A, B, . . . , C }), $\mathbf{Y}$ is a set of output tapes (e.g., $\mathbf{Y}$={ U, V, . . . , T }), $\mathbf{Z}$ is a set of internal variables (e.g., $\mathbf{Z}$={ a, b, . . . , v }), $\mathbf{P}$ is a flowchart constructed from statements of $\mathbf{L_0}$. We require also all exits of statements in flowchart to be attached to some statements, i.e., no pending exits are allowed (c.f. the case in Section 4). We also assume the flowchart to be connected. The execution starts from the first statement (marked by the label "→"). Program stops when it reaches a STOP statement.

Fig. 2 gives an example of a program which creates a new sorted tape (file) by merging sorted tapes A and B. The program has a bug: control from statement 7 is passed to statement 8 (instead of 10).

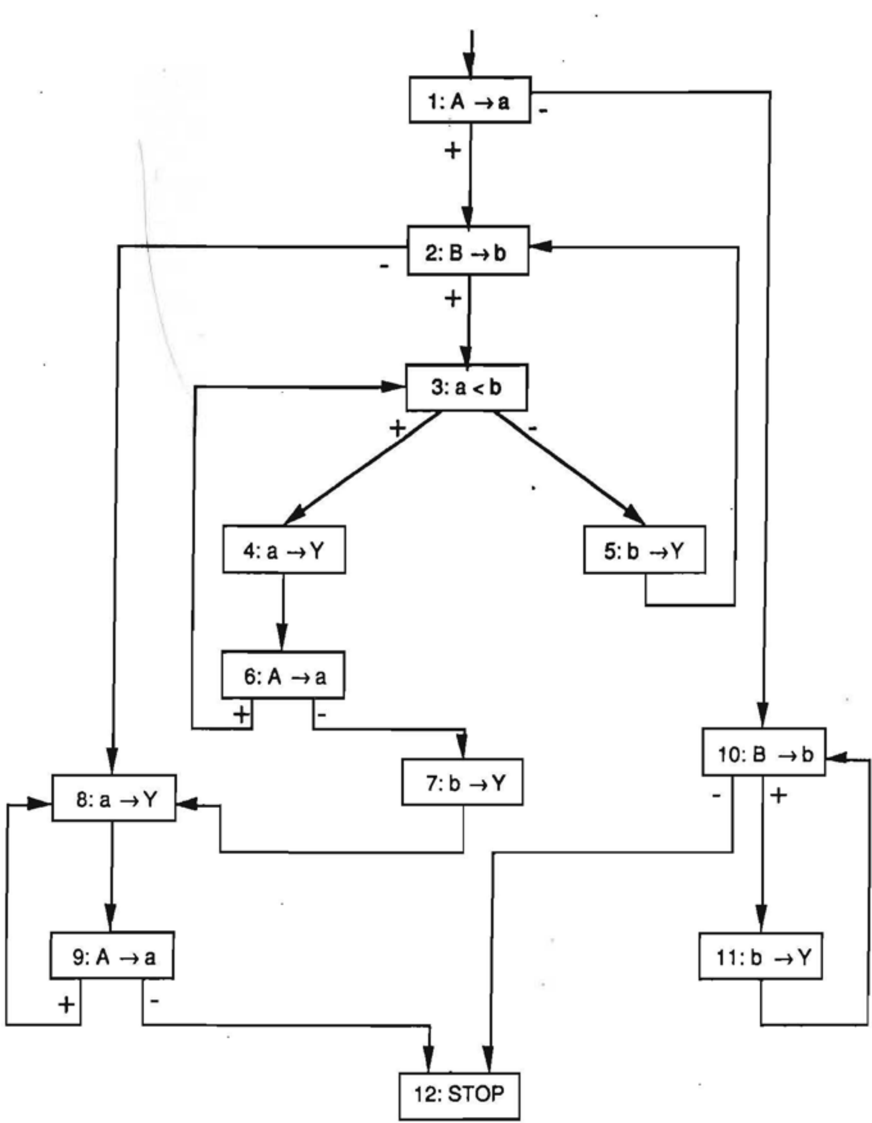


Fig. 2

By program *path* we understand a statement sequence $(k_1, k_2, \ldots, k_r)$, where each statement k_i has one of its exits ("+" or "-") fixed and this exit leads to the statement k_{i+1}, $i=1, 2, \ldots, r-1$.

The program in fig. 2 contains, for example, the path α=(1: A→a+, 2: B→b+, 3: a<b+, 4: a→Y, 6: A→a+) or simply α=(1+, 2+, 3+, 4, 6+), if only labels of statements are used.

If the path starts from the first statement of the program, we call it *initial path.* A path $\alpha=(k_1, k_2, \ldots, k_r)$is called a program *branch* if k_1 is a conditional statement (or the first statement of the program), $k_2, k_3, \ldots k_r$ are unconditional statements and the exit of k_r leads to a conditional statement (or a STOP statement). For example, the program in fig. 2 has branches (1+), (10+, 11), (1-), etc.

2.2 CTS Construction Problem

By *test* T for program P=($\mathbf{X}$, $\mathbf{Y}$, $\mathbf{Z}$, $\mathbf{P}$) we understand an association which associates a sequence of integers to each of the input tapes (i. e., to each element of set $\mathbf{X}$). Let us say that test T executes the branch if this branch is executed while running program P on test T. When the program in fig. 2 is run on the test A=(0), B=(1), the path (1+, 2+, 3+, 4, 6-, 7, 12) containing branches (1+), (2+), . . . is executed.

A test set is said to be a *complete test set* (CTS) for the given program if every *feasible* branch (i.e., branch executable by some test) is executed by some test of this set. For the program in fig. 2, for example, the following test set is complete: T1={ A=(0, 1), B=(2)}, T2={A=(6), B=(1, 2, 3)}, T3={ A=(2), B=(0, 2)}, T4={A=(1, 2, 3), B=(0)}, T5={A=( ), B=(0, 1, 2)}. It is easy to see that the bug in the program is found on this set.

Evidently for every program there exists a finite CTS. The main problem is to find this set.

THEOREM 1. *There is an algorithm for constructing a finite complete test set for every program in* L_0.

The proof will consist of several auxiliary assertions.

An important role in the proof will be played by systems of inequalities. At first let us introduce a slightly extended inequality relation $<(r)<$, where r=0, 1, . . . We say that $x < (r) < y$ if $y - x \geq r$. Now the rule of transivity is the following: $x < (r) < y \;\&\; y < (p) < z \rightarrow x < (r+p)< z$.

By *a system of inequalities* we understand the following system

$$\begin{array}{l} x_1 < (r_1) < y_1 \\ \ldots\ldots\ldots \\ x_n < (r_n) < y_n , \end{array}$$

where $x_1, \ldots, x_n, y_1, \ldots, y_n$ are variables or integer constants, for example,

$$\begin{array}{l} a < (0) < 3 \\ b < (2) < a \\ b < (5) < 3 \\ b < (3) < c \\ c < (0) < d \\ d < (0) < c \\ -4 < (1) < b \\ -4 < (2) < 3. \end{array}$$

We represent systems of inequalities also as *graphs*: vertices are labeled by variables

and constants of the system and an edge of the weight r is drawn from vertex y to vertex x if the system contains inequality x < (r) < y. So the previous example of inequality system corresponds to the graph in fig. 3.

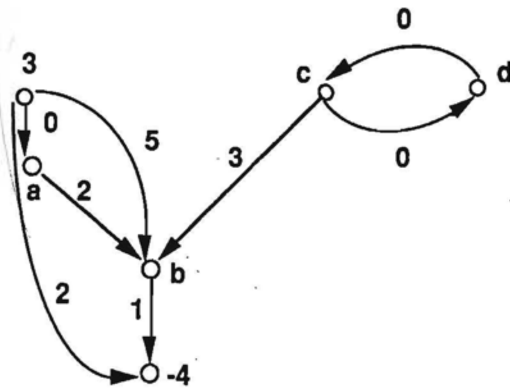


Fig. 3

Vertices labeled by variables are called *variable vertices* (vertices a, b, c, d in fig. 3), and vertices labeled by constants are called *constant vertices* (vertices 3 and -4).

Let us consider a directed path in the graph of inequality system. We define the weight of the path to be the sum of weights of the edges contained in the path.

Let us remark the following simple lemma.

LEMMA 1. *The inequality system N has a solution iff its graph G_N has the following properties:*

1. There is no cyclic path with the weight greater than 0.

2. The weight of every path leading from a constant vertex c_1 to a constant vertex c_2 is not greater than c_1-c_2.

The necessity is obvious. Let us prove the sufficiency. We build the solution by induction: at every induction step we assign constant value to some variable vertex of G_N in such a way that, first, the assigned constant satisfies all inequalities of the graph concerning the given vertex and, second, the replacement of the variable vertex by the constant vertex in the graph preserves validity of lemma conditions. To do this we need some additional notions. Let us say that vertex x is p units (p>0) greater (less) than vertex y if the maximum weight of paths leading from x to y (from y to x) is equal to p. Let us consider a variable vertex z in G_N with no constant value assigned in the previous steps. Let us find all constant vertices (including the ones created in the previous steps) with the paths leading to them from z.

Let $c_1^1, c_2^1, \ldots, c_k^1$ be the values of these vertices and $p_1^1, p_2^1, \ldots, p_k^1$ be numbers showing how many units the corresponding vertex is less than z. Similarly we find all constant vertices with the paths leading from them to z. Let $c_1^2, c_2^2, \ldots, c_l^2$ be their values and $p_1^2, p_2^2, \ldots p_l^2$ be numbers showing how much they are greater than z. Now let us consider numbers

$u_i^1=c_i^1+p_i^1$, i=1, 2, . . . , k, and

$u_i^2=c_i^2-p_i^2$, i=1, 2, . . . l.

Let u_1=max($u_1^1, u_2^1, \ldots, u_k^1$) and u_2=min($u_1^2, u_2^2, \ldots, u_l^2$). If the set { $u_1^1, u_2^1, \ldots, u_k^1$ } is empty (i.e., there are no paths from z to constant vertices), then let us assume $u_1=-\infty$. Similarly, if the set { $u_1^2, u_2^2, \ldots, u_l^2$ } is empty, then let $u_2=+\infty$. Now let us consider the interval [u_1, u_2].

If the lemma conditions (namely, condition 2) are valid for G_N after previous induction steps, then it is easy to see that the interval [u_1, u_2] is nonempty. It is also easy to see that every value of the variable z within the interval [u_1, u_2] satisfies inequalities of G_N with respect to the constants contained in G_N up to that moment (taking into account also transitivity). So let us choose any value from this interval and assign it to z. So vertex z now is a constant vertex. From the abovementioned it is clear that such replacement of variable vertex by constant vertex in G_N preserves the validity of the lemma conditions. By continuing the replacement of variable vertices by constant ones the same way we obtain the solution of the system of inequalities N.

This proves the lemma.

Let us remark that the proof of the lemma yields a simple algorithm for solving inequality systems.

Let P be a program in $\mathbf{L_0}$ and α=($k_1, k_2, \ldots, k_r$) an initial path in this program. Now let us define a system of inequalities N(α) corresponding to path α. N(α) will describe the feasibility conditions of path α. Let α_i denote the initial segment ($k_1, k_2, \ldots, k_i$) of path α, i=1, 2, . . . , r, and α_0 the empty initial segment. Let X be an input tape and t, u internal variables of program P. Let t_k, u_l be variables denoting the values of variables t, u after the execution of path α_{i-1}=($k_1, k_2, \ldots, k_{i-1}$) and X_s the last cell of tape X read during the execution (at the beginning the corresponding variables are t_0, u_0, X_0). Let c be a constant.

The system N(α) will be defined inductively: N(α_0), N(α_1), . . ., N(α_r)= N(α). Let us remember that internal variables are equal to 0 in the beginning. Therefore we define

$$N(\alpha_0) = \begin{cases} t_0=0 \\ u_0=0 \\ \ldots \end{cases}$$

Now let us assume that the system of inequalities N(α_{i-1}) is already defined. Then we define N(α_i) as a system obtained from N(α_{i-1}) by adding the following inequalities (in the sequel we use also standard inequality and equality relations $x \le y$, $x<y$, $x=y$ understanding by them $x<(0)<y$, $x<(1)<y$, $x<(0)<y$ & $y<(0)<x$ respectively):

(1) If k_i=(X→u-), then inequality X<0 is added. By inequality X<0 we code the fact that integer sequence on tape X is exhausted.

(2) If k_i=(X→u+) and N(α_{i-1}) does not contain inequality X<0 (i.e., no statement of type X→u- has been performed), then equality

$$u_{l+1}=X_{s+1}$$

is added. In this case new variables u_{l+1} and X_{s+1} are introduced which have the same sense for statement k_{i+1} as u_l and X_s have for k_i. If inequality X< 0 has already occured, then inequality 0< X is added in order to obtain a contradictory system.

(3) If k_i=(t→u) (or k_i=(c→u)), then equality

$$u_{l+1}=t_k \quad (\text{ or } u_{l+1}=c\ )$$

is added. A new variable u_{l+1} is again introduced in this case.

(4) If $k_i=(t<u+)$ (or $k_i=(c<u+)$ or $k_i=(t<c+)$), then inequality
$t_k<(1)<u_l$ (or $c<(1)<u_l$) or $t_k<(1)<c$) is added.

(5) If $k_i=(t<u-)$ (or $k_i=(c<u-)$ or $k_i=(t<c-)$), then inequality
$t_k>(0)>u_l$ (or $c>(0)>u_l$ or $t_k>(0)>c$) is added.

Let us give an example. For $\alpha^*=(1+, 2+, 3+)$ (see fig. 2) we have the inequality system

$$N(\alpha^*) = \begin{cases} a_0=0 \\ b_0=0 \\ a_1=A_1 \\ b_1=B_1 \\ a_1<(1)<b_1 \end{cases}$$

$N(\alpha^*)$ is represented as a graph in fig.4.

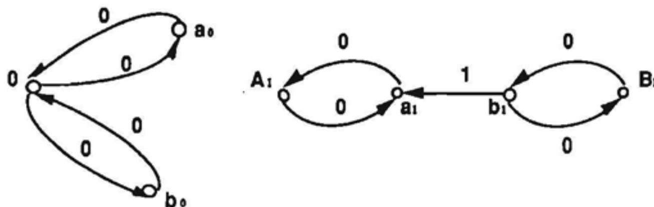


Fig.4

From the construction of $N(\alpha)$ there follows:

LEMMA 2. *The path α is feasible iff the system $N(\alpha)$ has an integer solution. Any solution of $N(\alpha)$ with respect to variables - cells of input tapes yields a test executing path α.*

Our aim is to reduce $N(\alpha)$ while preserving the existence (or the nonexistence) of the solution in such a way that there will be only a finite number of possible reduced systems for the given program P. This reduction relies on a variable exclusion method. Let us introduce some notations for this purpose.

Let us consider all the constants of program P (including 0). Let c_1 be the minimal and c_2 the maximal among these constants. Let $c_0 = c_2-c_1$. Let us consider an arbitrary system of inequalities N (e.g. $N(\alpha)$) where all constants are within the segment $[c_1, c_2]$ and weights of edges within $[0, c_0+1]$. Let y be a variable in system N. Now let us define the exclusion of variable y. We consider all the pairs x, z of variables and / or constants distinct from y for which there exist inequalities $x < (p_1) < y$ and $y < (p_2) < z$ in the system N. For each of these pairs we add a new inequality $x < (r) < z$ to N where $r=p_1+p_2$ if $p_1+p_2 \leq c_0+1$ and $r=c_0+1$ if $p_1+p_2 > c_0+1$. Then we delete all inequalities containing y from N. If N contains inequality of the type $y<(p)<y$ with $p > 0$, then this inequality is replaced by some standard contradicting inequality, e.g., $0 < (1) < 0$ (because the new system must have no solutions). So obtained inequality system is denoted by N'. From the construction of N' there follows an assertion:

The conditions for solution existence from Lemma 1 hold for inequality system N iff these conditions hold for inequality system N'. In other words, *inequality system N has a solution iff N' has a solution.*

Now let us return to inequality system $N(\alpha)$. Let program P have input tapes A, B,

. . . and internal variables t, u, ... Then the system $N(\alpha)$ contains, in general, variables A, B, . . . ; $A_1, A_2, \ldots, A_d$; $B_1, B_2, \ldots, B_e$; . . . ; $t_1, t_2, \ldots, t_f$; $u_1, u_2, \ldots, u_g$; . . . . Let us remember that internal variables with maximal subscripts t_f, u_g, . . . denote values of internal variables t, u, . . . after the execution of path α. These variables t_f, u_g, . . . , as well as variables A, B, . . . denoting input files are called active variables, the other ones - inactive. For example, the system $N(\alpha^*)$ from the previous example has active variables a_1, b_1. Now let us exclude, one after another, all inactive variables from $N(\alpha)$. It is easy to see that the order of variable exclusion does not affect the resulting system. Thus, we obtain a new system of inequalities containing only active variables. Then we drop all subscripts of the variables in it. The resulting system is denoted by $S(\alpha)$ and called a *program state after the execution of path* α.

Informally the state describes relations between current values of internal variables. The state corresponding to the path α^*=(1+, 2+, 3+) from the previous example, as it can be easily deduced from the inequality system $N(\alpha^*)$, is

$S(\alpha^*)$={ a<(1)<b.

From the assertion about variable exclusion and state construction there follows

LEMMA 3. *A path α is feasible (i.e., system $N(\alpha)$ has an integer solution) iff the state $S(\alpha)$ is consistent (i.e., $S(\alpha)$ has an integer solution as a system of inequalities).*

The system of inequalities containing only internal variables and constants of program P and having no weights of inequalities r greater than c_0+1 is called *state* of the program P. Like every system of inequalities a state can be represented by a graph. Two states will be called equal if the corresponding graphs are isomorphic (as graphs with labeled vertices and edges). It is easy to see that every program P has a finite number of distinct states. This fact together with the next lemma will play the main role in the proof of Theorem 1.

Now we need to generalize slightly the notion of the system of inequalities $N(\alpha)$ for path. At first there will be no longer the requirement for a path α=($k_1, k_2, \ldots, k_r$) to be initial. Further, we allow to have arbitrary state σ of program P as an initial inequality system for construction. Under these conditions we define the system

$N(\sigma,\alpha)$

the following way. $N(\sigma,\alpha_0)$ is the same initial inequality system σ with only zero subscripts added to internal variables: $t_0, u_0, \ldots$.

Further, $N(\sigma,\alpha_i)$ is defined from $N(\sigma,\alpha_{i-1})$ and statement k_i just as before. For example, if σ={a<(1)<b and α=(4, 6+, 3+), then

$N(\sigma,\alpha_0)$={a_0<(1)<b_0,
$N(\sigma,\alpha_1)$={a_0<(1)<b_0 (output statement adds nothing),
$N(\sigma,\alpha_2)$={a_0<(1)<b_0 , $a_1=A_1$,
$N(\sigma,\alpha)=N(\sigma,\alpha_3)$={$a_0$<(1)<$b_0$, $a_1=A_1$, a_0<(1)<b_0 .

Just as before we define state $S(\sigma,\alpha)$ corresponding to system $N(\sigma,\alpha)$, the state is obtained by excluding inactive variables from $N(\sigma,\alpha)$ the same way. For example, the state

$S(\sigma,\alpha)$={a<(1)<b

corresponds to the system $N(\sigma,\alpha)$ from the previous example. Let us note that this time state $S(\sigma,\alpha)$ occurs to be equal to σ. It has the following simple graph depicted in fig. 5.

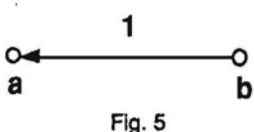


Fig. 5

A path β is said to be a continuation of a path α if the exit of the last statement of the path α leads to the first statement of the path β. The concatenation of paths α and β is denoted by α+β.

LEMMA 4. *Let σ be a state of program P, α a path and β a continuation of path α. Then*

$$S(\sigma,\alpha+\beta)=S(S(\sigma,\alpha), \beta).$$

To prove the lemma we consider systems of inequalities N(σ,α+β), N(σ,α) and N(S(σ,α), β). By excluding inactive variables we obtain states S(σ,α+β), S(σ,α) and S(S(σ,α), β) from them. Let us remember that the order of exclusion of inactive variables does not affect the result. Therefore while constructing S(σ,α+β) from N(σ,α+β) we can exclude inactive variables in the starting period just in the same order as when constructing S(σ,α) from N(σ,α). It means that while constructing S(σ,α+β) we obtain S(σ,α) as an intermediate result and the construction of S(S(σ,α), β) from N(S(σ,α), β) follows just after that. In this consideration we have essentially used two facts; first, inactive variables in system N(σ,α) are also inactive in N(σ,α+β) and, second, inequalities in N(σ,α+β) generated by the path (i.e., the continuation of N(σ,α) up to N(σ,α+β)) do not contain inactive variables of N(σ,α). It means that the exclusion of these inactive variables does not affect the continuation of N(σ,α) up to N(σ,α+β). All this becomes completely clear if we represent systems of inequalities and the exclusion process as graphs.

This completes the proof of Lemma 4.

Now let us construct the reachability graph for program P. Vertices of this graph are labeled by pairs (n,S), where n is a statement label and S a state of P. There will be as many edges issuing and with the same labels from the vertex (n,S) as from the statement n in P. Simultaneously with the vertex we also build the edges issuing from it (for the moment they are pending). The construction of the graph will be by induction. The initial vertex of the reachability graph will be the pair (n_0,S_0), where $n_0=1$ and S_0 is the initial state of program P: t=0, u=0, ... Edges issuing from this vertex will be pending for a while.

Let us assume that part of reachability graph has been constructed. Edges issuing from its vertices can be in three different states:

(1) an edge can be pending,

(2) an edge can be joined to a vertex,

(3) an edge can be forbidden (the emergence of forbidden edges will be explained further).

Only the pending ones will be of interest. So we choose a vertex (n_i,S_i) with a pending edge labeled by ε issuing from it (ε belongs to {+, -, e}). Let γ_i denote a path consisting of the sole statement n_i with exit ε : $n_i\varepsilon$. Let us build the state $S_j=S(S_i,\gamma_i)$. Two cases are possible:

(1) the state S_j is contradictory, i.e., it has no solution as a system of inequalities; in this case the exit ε from the vertex (n_i,S_i) is said to be forbidden (for example, the edge

is marked by special label "X"). (Let us remind that our notion of directed graph allows pending edges in it),

(2) the state S is consistent. Let n_j be the label of statement entered by exit ε of the statement n_i. Let us consider the pair (n_j,S_j). Again two cases are possible:

(a) the vertex (n_j,S_j) exists in the part of reachability graph already constructed; in this case we join the edge ε from vertex (n_i,S_i) to vertex (n_j,S_j),

(b) the vertex (n_j,S_j) does not exist in the part already constructed; in this case we build a new vertex (n_j,S_j) together with all the pending edges issuing from it and then join the edge labeled by ε from vertex (n_i,S_i) to the new vertex.

The described procedure is continued until we obtain a graph with no unmarked pending edges. Since the program P has a finite number of different states the before mentioned procedure will stop after a finite number of steps. The graph obtained as a result of this procedure we call *reachability graph* of the program P.

This graph has several important properties. Let us consider a path

$$v=((n_0,S_0)\varepsilon_0,\ (n_1,S_1)\varepsilon_1,\ \ldots,(n_r,S_r)\varepsilon_r)$$

in this graph starting from the initial vertex. Such a path will be called an *initial path.* Such a path may not contain forbidden edges. It means that edge ε_r leads to some vertex, say (n_{r+1},S_{r+1}).

We shall say that the path v is *feasible* if there is a test T such that the program P executes the path

$$\alpha=(n_0\varepsilon_0,\ n_1\varepsilon_1,\ \ldots,\ n_r\varepsilon_r)$$

and passes the state sequence

$$S_1,\ S_2,\ldots,\ S_{r+1} \text{ on this test.}$$

From the construction of reachability graph and Lemma 4 there follow equalities $S_i=S((n_0\varepsilon_0,\ n_1\varepsilon_1,\ \ldots,n_{i-1}\varepsilon_{i-1}))$ for $i=1,\ 2,\ \ldots,r+1$. The state S_{r+1} is consistent by construction. Thus the state $S((n_0\varepsilon_0,\ n_1\varepsilon_1,\ \ldots,\ n_r\varepsilon_r))$ is also consistent and the path α in P is feasible by Lemma 3. This proves the following

LEMMA 5. *Every initial path in reachability graph is feasible.*

Let v be the beforementioned path in rechability graph. In this case the path

$$\alpha=(n_0\varepsilon_0,\ n_1\varepsilon_1,\ \ldots,\ n_r\varepsilon_r)$$

in program P will be said to be the *projection* of path v .

LEMMA 6. *An initial path α in program P is feasible iff there is an initial path v in rechability graph whose projection is α.*

The sufficiency of lemma condition follows directly from Lemma 5. Let us prove the necessity. Let

$$\alpha=(n_0\varepsilon_0,\ n_1\varepsilon_1,\ \ldots,\ n_r\varepsilon_r)$$

be an initial path in program P. Let us consider the sequence of current states $S_0=S(\alpha_0)$, $S_1=S(\alpha_1)$, $S_2=S(\alpha_2)$, $\ldots$,$S_{r+1}=S(\alpha_{r+1})=S(\alpha)$, where $\alpha_i=(n_0\varepsilon_0,\ n_1\varepsilon_1,\ \ldots,\ n_{i-1}\varepsilon_{i-1})$. It follows from Lemma 3 that the path α is feasible iff the states S_0, S_1, . . ., S_{r+1} are consistent. Further it follows from Lemma 4 that the states S_0, S_1, . . ., S_{r+1} can be obtained also in a different manner:

$$S_1=S(S_0,n_0\varepsilon_0),\ S_2=S(S_0,\ n_1\varepsilon_1),\ \ldots,\ S_{r+1}=S(S_r,n_r\varepsilon_r),$$

i.e., by constructing the new state from the previous one and the current statement.

Hence, and from the construction of reachability graph, it follows that feasibility of path α implies the existence of path

$$((n_0,S_0)\varepsilon_0, (n_1,S_1)\varepsilon_1, \ldots, (n_r,S_r)\varepsilon_r)$$

in the reachability graph whose projection is path α.
This proves the lemma.

Let U be a set of initial paths in the reachability graph. U is said to be a *complete path set* if it contains all edges of the graph. Let us denote by pr_U a set of paths in the program P obtained by taking projections of paths in U.
From Lemma 6 there follows an obvious

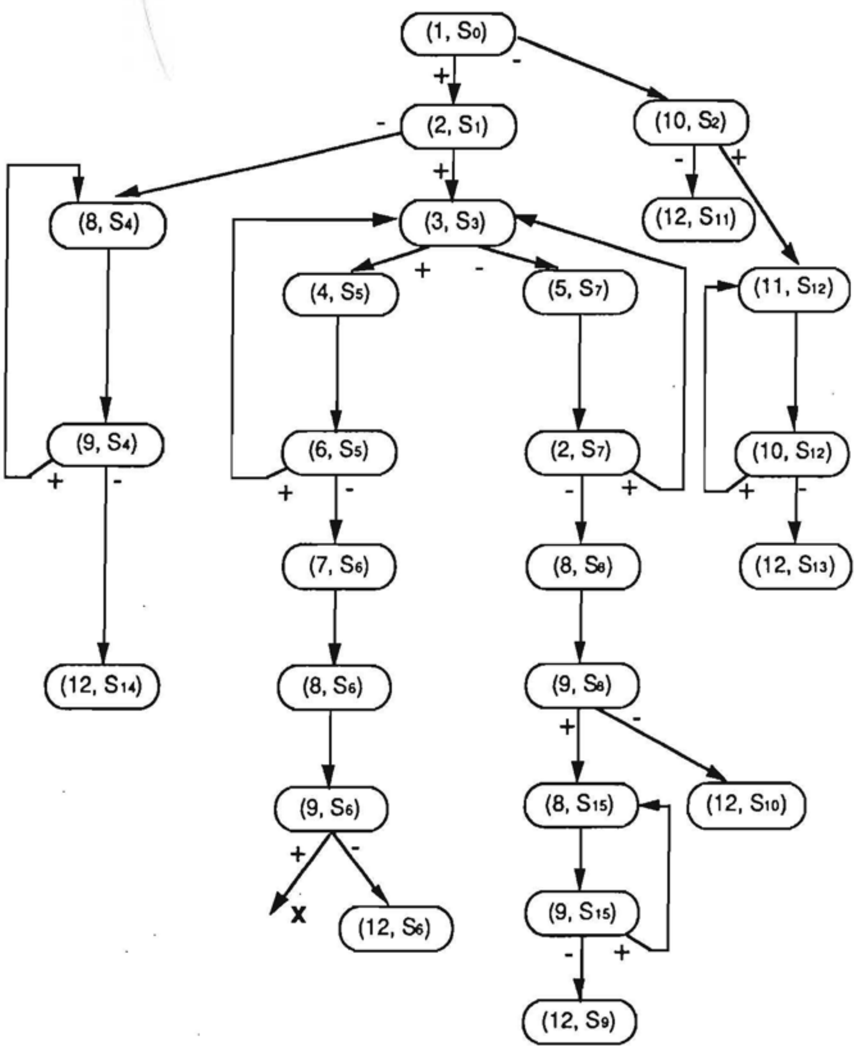


Fig. 6.1

COROLLARY. *Let U be a complete path set of the reachability graph of program P. Then the set of program P paths pr_U contains all feasible branches of program P.*

Hence follows the algorithm constructing CTS for a program P:
1. Construct reachability graph G for the program P. Fig. 6.1 and 6.2 show the reachability graph for the program in fig. 2.

$S_0=$ {a=0, b=0}
$S_1=$ {b=0}
$S_2=$ {A<0, a=0, b=0}
$S_3=$ {}
$S_4=$ {B<0, b=0}
$S_5=$ {a<b}
$S_6=$ {a<b, A<0}
$S_7=$ {b≤a}
$S_8=$ {b≤a, B<0}
$S_9=$ {B<0, A<0}
$S_{10}=$ {b≤a, B<0, A<0}
$S_{11}=$ {A<0, a=0, b=0, B<0}
$S_{12}=$ {A<0, a=0}
$S_{13}=$ {A<0, B<0, a=0}
$S_{14}=$ {A<0, B<0, b=0}
$S_{15}=$ {B<0}

Fig.6.2

2. Construct a complete path set U for graph G consisting of finite paths. It is clear that there exists an efficient algorithm for finding such a path set. Henceforth we shall construct this path set the following way: we go along the "+" branches until the vertices repeat, then we interrupt the path and start a new one repeating the previous path up to the last (i.e., the first from bottom) "-" branch, select this "-" branch and again proceed along "+" branches, etc. Fig 7. shows the covering of reachability graph for the abovementioned program.

p_1=(1,S_0)+, (2,S_1)+, (3,S_3)+, (4,S_5)+, (6,S_5)+, (3,S_3)
p_2=(1,S_0)+, (2,S_1)+, (3,S_3)+, (4,S_5)+, (6,S_5)-, (7,S_6), (8,S_6), (9,S_6)-, (12,S_6)
p_3=(1,S_0)+, (2,S_1)+, (3,S_3)-, (5,S_7), (2,S_7)+, (3,S_3)
p_4=(1,S_0)+, (2,S_1)+, (3,S_3)-, (5,S_7), (2,S_7)-, (8,S_8), (9,S_8)+, (8,S_{15}), (9,S_{15})+, (8,S_{15})
p_5=(1,S_0)+, (2,S_1)+, (3,S_3)-, (5,S_7), (2,S_7)-, (8,S_8), (9,S_8)+, (8,S_{15}), (9,S_{15})+, (8,S_{15}), (9,S_{15})-, (12,S_{12})
p_6=(1,S_0)+, (2,S_1)+, (3,S_3)-, (5,S_7), (2,S_7)-, (8,S_8), (9,S_8)-, (12,S_9),
p_7=(1,S_0)+, (2,S_1)-, (8,S_4), (9,S_4)+, (8,S_4)
p_8=(1,S_0)+, (2,S_1)-, (8,S_4), (9,S_4)-, (12,S_{14})
p_9=(1,S_0)-, (10,S_2)+, (11,S_{12}), (10,S_{12})+, (11,S_{12})
p_{10}=(1,S_0)-, (10,S_2)+, (11,S_{12}), (10,S_{12})-, (12,S_{13})
p_{11}=(1,S_0)-, (10,S_2)-, (12,S_{11})

Fig. 7

3. Take pr_U. For every $\alpha \in$ pr_U construct the inequality system $N(\alpha)$ and find its solution with respect to variables-cells of input tapes. This solution forms the test T_α.

The test set

$$T = \{ T_\alpha \mid \alpha \in pr_U \}$$

is obtained as a result.

Test set corresponding to the covering in fig. 7 is depicted in fig. 8.

$T_1 = \{ A = (0,1),\quad B = (1)\}$
$T_2 = \{ A = (0),\quad B = (1)\}$
$T_3 = \{ A = (1),\quad B = (0,1)\}$
$T_4 = \{ A = (1,2,3),\quad B = (0)\}$
$T_5 = \{ A = (1,2),\quad B = (0)\}$
$T_6 = \{ A = (1),\quad B = (0)$
$T_7 = \{ A = (0,1),\quad B = ( )\}$
$T_8 = \{ A = (0),\quad B = ( )\}$
$T_9 = \{ A = ( ),\quad B = (0)\}$
$T_{10} = \{ A = ( ),\quad B = (0)\}$
$T_{11} = \{ A = ( ),\quad B = ( )\}$

Fig. 8

It is clear that test T_2 reveals the bug yielding wrong result Y=(0, 1, 0).

It follows from the beforementioned that **T** is a complete test set for program P.
This completes the proof of the Theorem.

2.3 Termination Problem

Some problems of program static analysis are closely related to the construction of CTS, reachability problem is one of them. The problem is to find out whether all program branches are feasible (reachable). It is easy to see that this problem is a special case of CTS problem and therefore no more attention is paid to it.

The second important problem is termination problem. The problem is to find out for a program whether it terminates on all input data selections. If there exist input data where the program does not stop, the program is said to be *nonterminating.* The decidability of reachability problem for a class of programs does not imply the decidability of termination at all. Therefore the following theorem arouses some interest.

<u>**THEOREM 2.**</u> *There is an algorithm which determines for every program in* L_0 *whether the program is nonterminating.*

To prove the theorem we use substantially the notion of reachability graph from the proof of the previous theorem.

Path $\beta=((n_1,S_1)\varepsilon_1, \ldots, (n_k,S_k)\varepsilon_k)$ in the reachability graph G is said to be *closed* if

(1) β is a cyclic path, i.e., the exit ε_k leads to the vertex (n_1, S_1),

(2) β contains no input statement with "+" exit, i.e., if n_i is statement $X \rightarrow t$, then ε_i is "-".

<u>**LEMMA 7**</u>. *Program P does not terminate on a test T iff a vertex of a closed path in the*

reachability graph can be reached on this test. Program P is nonterminating iff there is a closed path in its reachability graph.

At first let us assume that there is a test T on which the program P does not terminate. It means that the execution of P on T creates an infinite path

$v=(n_1\varepsilon_1, n_2\varepsilon_2, \ldots)$.

Since the test T is finite, there is l such that l-tail of path v

$\delta=(n_l\varepsilon_l, n_{l+1}\varepsilon_{l+1}, \ldots)$

contains no more input statements with "+" exit. Since the set of program states is finite and the number of statements is finite, the path δ certainly will contain a segment

$(n_j\varepsilon_j, \ldots, n_{j+u}\varepsilon_{j+u})$

such that ε_{j+u} leads to the statement n_j and states S_j and S_{j+u+1} are equal. It means that there is a closed path, namely,

$((n_j,S_j)\varepsilon_j, \ldots, (n_{j+u}, S_{j+u})\varepsilon_{j+u})$

in the reachability graph reached on the test T. This proves the necessity of lemma condition.

Let us prove sufficiency. Let us assume that there is a closed path

$\beta=((n_1,S_1)\varepsilon_1, \ldots, (n_k,S_k)\varepsilon_k)$

in the reachability graph G. Let α be an initial path leading to the vertex (n_1,S_1). Let us consider path αββ...β with the segment β repeated v times, v - "large enough". This path, like every path in a reachability graph, is feasible. Since the segment ββ...β contains no input statements with "+" exit, the values of internal variables will begin to repeat. It means that, if the program executes path αββ...β on some test T, then it will continue to repeat the segment β on the same test, i.e., it will loop forever. Hence, by the way, follows that a closed path β has the property that conditional statements contained in the path have only one of their exits executable, namely, the one contained in the path β. In other words, there can be no paths γ branching off the closed path in the reachability graph. Actually, were such a path γ, then, taking into consideration that all paths are feasible in reachability graph the test forcing the path αββ . . . ββ would also force the path αββ...βγ. This yields a contradiction. It means that the program loops forever on every test where some vertex of the closed path is reached. It proves the sufficiency of lemma conditions.

The condition of nontermination used in the lemma is algorithmically decidable. This proves the theorem.

3 Efficient Algorithms for CTS Construction

The proof of Theorem 1 gives us an algorithm for CTS construction which is not very efficient, especially because of the size of reachability graph. To reduce the size of this graph we introduce two notions: essentially located statements and essential variables.

A set of program statements is selected in such a way that every program loop contains at least one statement from this set.The first statement of the program and STOP statements are also included in this set. We call the statements from this set *essentially located statements* (ELS's). Our intent is to keep the set of ELS's as small as possible, therefore, if several loops have a common part, ELS is selected from this

common part. In the program of fig. 2, for example, statements 1, 3, 9, 10, 12 form a set of ELS's.

Associated with every ELS there is a list of variables called *essential variables associated with the ELS.* An internal variable t is said to be an essential variable for a certain ELS if there exists a path beginning with the ELS such that the value possessed by the variable t immediately before the execution of the ELS is used unchanged in some comparison statement of the path.The use of unchanged value in comparison statement means that either the variable t is contained in some comparison statement (e.g., $t<9$) of the path before the new value is assigned to t or the unchanged value of t is assigned to some other variable u which, in turn, is used unchanged in a comparison statement.

There are several ways to find out whether the given variable is essential for the given ELS. We give an algorithm which is based on reverse analysis of program path from the end to the beginning. When traversing a path in a reverse order, a set of essential variables V is formed according to the following rules depending on the current statement K:

(1) if K is $t<v$ (both "+" and "-" exit), then $V:= V\cup\{t, v\}$;
(2) if K is $t<c$ (both exits), then $V:= V\cup\{t\}$;
(3) if K is $X\rightarrow t+$, then $V:= V\setminus\{t\}$;
(4) if K is $X\rightarrow t-$, then V is not changed;
(5) if K is $t\rightarrow v$, then $V:=$ if $v\in V$ then $V\cup\{t\}\setminus\{v\}$ else V;
(6) if K is $c\rightarrow v$, then $V:= V\setminus\{v\}$.

Further we form a graph for program P with ELS's as vertices and program paths e_i^j from one ELS to another as edges. Each vertex n has a set of essential variables V_n ascribed, initially all V_n are empty. Each vertex has also a status assuming one of the three values: not visited, active, inactive. Initially all vertices except those corresponding to STOP statements are not visited, STOP statements are marked active. On each step of the algorithm an active vertex n is selected, it is marked inactive and all edges e_i^j entering it are traversed as program paths in the reverse order as described before (V_n is taken as the initial value of V). When another ELS m is reached in the reverse analysis the resulting value of V is added to V_m. If V_m is actually increased, the status of m is set to active (also in the case when n=m). If the status of m was 'not visited', it is set to active anyway. Algorithm proceeds until all vertices are inactive. If the situation occurs where all vertices are either inactive or not visited, one of the not visited vertices is made active. The resulting values of V_n are the sets of essential variables for each of the ELS's. The termination of the algorithm is guaranteed by the monotonity of V_n for all n.

Of course, the feasibility of paths is not taken into account. In the program example considered, statement 1 has no essential variables because a and b are given new values from input tapes before using them. Statement 3 obviously has a and b as essential variables because the statement itself is a comparison statement using them. Statements 9, 10, 12 have no essential variables.

After these preparations a *reduced reachability graph* is constructed. Its construction is similar to that of the reachability graph. The main difference is that vertices correspond only to ELS's, other statements are not included. For each of the ELS's we build a set of all paths in the program starting with it and leading to some

other ELS. These paths, let them be $e^i_1, e^i_2, \ldots, e^i_r$ for ELS i, will play the role of edges in reachability graph construction. The choice of ELS's guarantees us the boundedness of this set for every ELS. The other difference is that when constructing a state for the given ELS we exclude from the corresponding system of inequalities also those internal variables which are not essential for this ELS. Let us remark that formal variables A, B,... used to code the exhaustion of input tape (e.g., A<0) are retained in state.

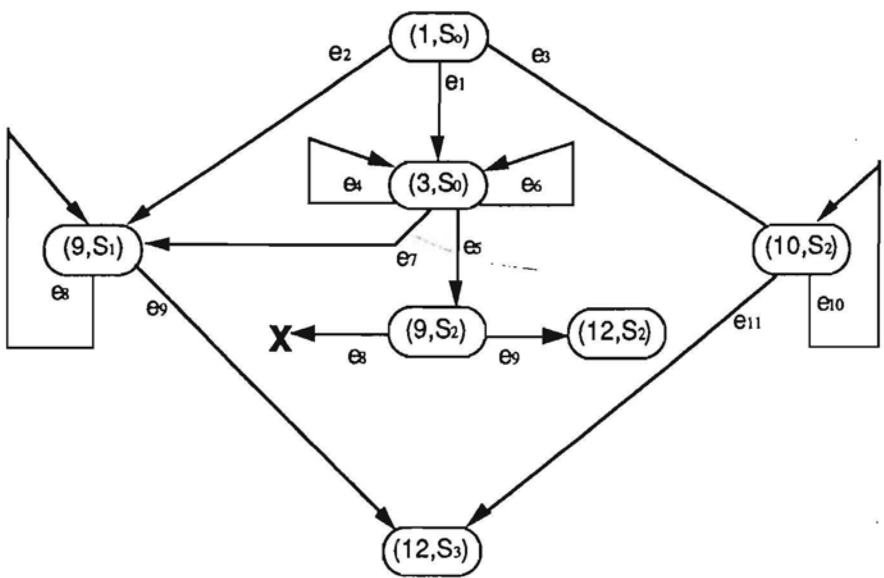
Likewise for reachability graph, the construction starts from the first statement of the program (which is ELS by definition) and empty state. For the given ELS i and state S_j, we consider the paths e^i_k from this ELS one after another. The system of inequalities $N(S_j,e^i_k)$ and state $S(S_j,e^i_k)$ (in the new sense with respect to ELS reached by e^i_k) are constructed for each path. If the path is infeasible (i.e., $S(S_j,e^i_k)$ is contradictory), it is labeled by X. Otherwise we check whether ELS i' reached by e^i_k and state $S(S_j,e^i_k)$ form a vertex already in the reduced reachability graph and join edge e^i_k to the existing vertex or build a new vertex respectively.

We also have to change the definition of path projection in reduced reachability graph, replacing every edge e^i_k by the corresponding sequence of program statements.

It can be shown that Lemmas 5 and 6 from Theorem 1 hold also for reduced reachability graph (the proof will not be given here).

The construction of CTS using reduced reachability graph is similar to the previous case. A more economical covering principle can be used where a path traverses all loops at the given vertex once and then proceeds further. For nearly all real programs the reduced reachabilty graph is considerably less than full reachability graph and thus completely outweighs some additional efforts to build it.

Now let us return to our example. ELS's and their essential variables had



$S_0 = \{$ $S_1 = \{ B<0$ $S_2 = \{ A<0$ $S_3 = \{ A<0\ B<0$

Fig. 9.

already been mentioned. Paths leading from one ELS to another are the following:

from 1	$e_1=(1+, 2+)$	to 3 ,
	$e_2=(1+, 2-, 8)$	to 9 ,
	$e_3=(1-)$	to 10 ,
from 3	$e_4=(3+, 4, 6+)$	to 3 ,
	$e_5=(3+, 4, 6-, 7, 8)$	to 9 ,
	$e_6=(3-, 5, 2+)$	to 3 ,
	$e_7=(3-, 5, 2-, 8)$	to 9 ,
from 9	$e_8=(9+, 8)$	to 9 ,
	$e_9=(9-)$	to 12 ,
from10	$e_{10}=(10+, 11)$	to 10,
	$e_{11}=(10 -)$	to 12

The reduced reachability graph is shown in fig.9, its covering in fig.10 and the corresponding CTS in fig.11. The bug is detected by T_1. It can be seen that states in fact contain no internal variables, for they don't affect the feasibility of paths in this simple program (c.f., in fact, surplus states S_{10}, S_{11}, S_{12}, S_{13}, . . . in fig.6.). The test set is also reduced but it still detects the bug.

$p_1 = (1, S_0)\ e_1, (3, S_0)\ e_4, (3, S_0)\ e_6, (3, S_0)\ e_5, (9, S_2)\ e_9, (12, S_2)$
$p_2 = (1, S_0)\ e_2, (9, S_1)\ e_8, (9, S_1)\ e_9, (12, S_3)$
$p_3 = (1, S_0)\ e_3, (10, S_2)\ e_{10}, (10, S_2)\ e_{11}, (12, S_3)$
$p_4 = (1, S_0)\ e_1, (3, S_0)\ e_7, (9, S_1)\ e_9, (12, S_3)$

Fig. 10

$T_1 = \{ A = (0,1), B = (1,2)\}$
$T_2 = \{ A = (0,1), B = ( )\}$
$T_3 = \{ A = ( ), B = (0)\}$
$T_4 = \{ A = (0), B = (0)\}$

Fig. 11

The reduced reachability graph can also be used for termination analysis of programs described in the previous section. We just note that every cyclic path will certainly contain some ELS and therefore will be present in the reduced reachability graph (in most cases as loop with one vertex in it).

There can be some further improvements of the algorithm for constructing CTS. At first let us remark the simple fact that for a program with all feasible paths we can simply construct its covering by paths and solve the corresponding inequality system by the method described. If it is not completely so, we start to construct the reduced reachability graph for the part of the program not traversable so simply and we look at every step of its construction (i.e., adding a path from one ELS to another) whether all branches have been covered. So with all statements reachable, usually only a small part of reachability graph is to be constructed. Both the original and improved algorithms are obviously exponential with respect to the size of the program in the worst case. Therefore no theoretical complexity analysis of the algorithms is given here. Nevertheless the performance of the algorithm described in this section is quite acceptable, for most real programs the numbers of steps required is nearly linear with

respect to the size. The practical aspects of CTS construction will be covered more thoroughly in [17].

There is another aspect of optimality, namely, the optimality of CTS obtained. It is reasonable to minimize the number of tests or the total size. Here the main issue is to find the optimal (with respect to the criterion selected) covering of the reduced reachability graph. Obviously it is very difficult to find the absolute optimum, nevertheless, algorithms yielding nearly optimal covering can be devised. The covering proposed in this section (traversing all the loops at the given vertex once and then proceeding further) is clearly oriented towards minimizing the number of tests.

4 Conditional Programs and Programs with Preconditions

In previous sections we have discussed only programs without pending exits, i.e, there was a requirement that every exit of a statement should be attached to some other statement. In the sequel such programs will be called *closed programs*. In this section we drop the requirement and consider also programs with pending exits, i.e., exits not attached to other statements. We shall say such exits to be *forbidden* exits and programs with forbidden exits to be *conditional programs*. Conditional programs offer us some new possibilities. By means of forbidden exits we can specify conditions on input data. Fig. 12 shows us a program for merging two nondecreasing files which in addition check whether the input files are really nondecreasing. To describe formally the meaning of such checks we introduce the notion of a correct test. A test is said to be *correct* if the program running on this test never reaches a forbidden exit. It is easy to see that for the program in fig. 12 only nondecreasing input files serve as correct tests.

In the case of a conditional program a test set will be called a *correct complete test set* if

(1) the test set contains only correct tests,

(2) all program branches executable on correct tests are executed on tests of this set.

It is clear that the construction of a correct complete test set is more complicated than the construction of usual CTS. Nevertheless there holds

THEOREM 3. *There is an algorithm constructig a finite correct complete test set for every conditional program in* $\mathbf{L_0}$.

To prove the theorem we consider an arbitrary program in $\mathbf{L_0}$ and its reachability graph. The definition of reachability graph for a conditional program is similar to that for a closed program. Let us remember that we already had a kind of forbidden edges when constructing the reachability graph, namely, we said that an edge ε from a vertex (n_i, S_i) is forbidden if the state $S(S_i,n_i\varepsilon)$ is contradictory, i.e., the exit ε of the statement n_i is infeasible in the state S_i. We call these forbidden edges the forbidden edges of the first type. Reachability graph for a conditional program will also have forbidden edges of the second type: we say that an edge ε from a vertex (n_i, S_i) is also forbidden in the case when the exit ε from the statement n_i is forbidden. There are no other differences in the construction of reachability graph for a conditional program. Lemma 6 holds true also for this case.

The main problem in the construction of correct CTS is to prevent the constructed tests from generating paths in the reachability graph leading to the forbidden edges of

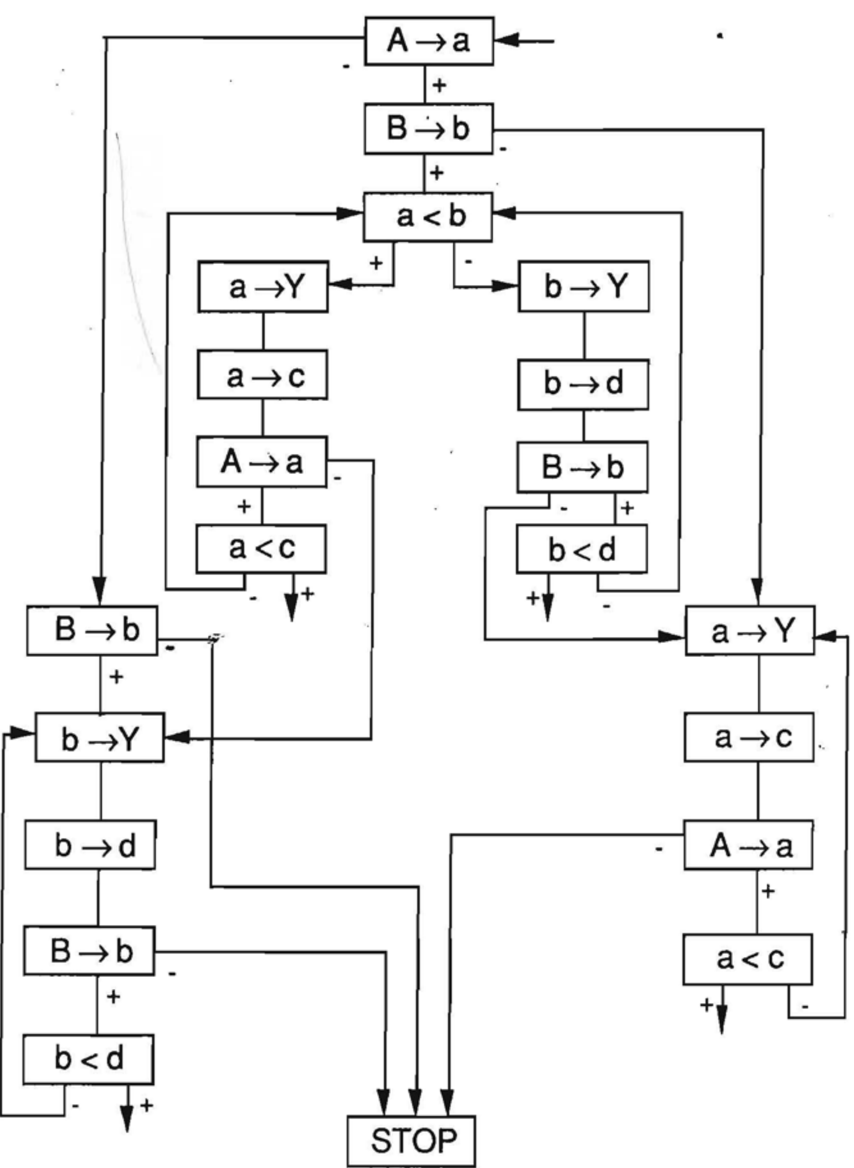


Fig. 12

the second type. To tackle the problem we remark the following facts. Lemma 7 can be generalized to conditional programs without difficulties. Hence follows that every correct test corresponds in the reachability graph to a path leading either to a STOP-vertex or to a vertex belonging to a closed path. On the other hand, if a STOP-vertex or a vertex in a closed path in the reachability graph is reached on some test, the test certainly is a correct one.

Now let us consider the vertices in the reachability graph from which it is impossible to reach either a STOP-vertex or a vertex in a closed path. We call these vertices the forbidden ones. It follows from the abovementioned that all program branches executable by correct tests belong to that part of reachability graph which remains when we delete all forbidden vertices (together with incoming edges).

Now let us delete all forbidden vertices from the reachability graph and call edges leading to them the forbidden edges of the third type. The graph so obtained will be called the abridged reachability graph. From the abovementioned there follows that all program branches executable by correct tests (and only such branches) belong to the abridged reachability graph. Evidently, all permitted edges of reachability graph can be covered by a finite set of paths where each of the paths either ends with a STOP-vertex or reaches a closed path. Tests corresponding to these paths will form a correct CTS.

This proves the theorem.

Now let us define programs with preconditions. To do this we consider predicates defined on a finite sequence of integers. We call such predicates tape conditions. We say that the value X^0 of tape X satisfies a tape condition S if $S(X^0)$=true.

Assume a program P with input tapes A, B, ... ,C and tape conditions $S_A, S_B, \ldots, S_C$ respectively be given. Let us call such a program to be a *program with preconditions* $S_A, S_B, \ldots, S_C$. For example, if we say that P is a program for merging two sorted tapes A and B (fig. 2), then in fact we assert that P is a program with preconditions S_A and S_B where both $S_A(X)$ and $S_B(X)$ are true if and only if the sequence $X = (x_1, x_2, \ldots, x_n)$ is nondecreasing.

We say that the test

$$A=A^0, B=B^0, \ldots, C=C^0$$

is a *correct test* for program P with preconditions $S_A, S_B, \ldots, S_C$ if

(1) the test satisfies the tape conditions $S_A, S_B, \ldots, S_C$,

(2) it is a correct test for P without preconditions (if P is a conditional program).

A test set T will be called a *correct complete test set* for program P with preconditions $S_A, S_B, \ldots, S_C$ if

(1) T consists only of correct tests,

(2) every branch of P executable by a correct test is executed on some test from T.

The question arises for what kind of preconditions the CTS problem is still solvable algorithmicaly for programs in $\mathbf{L_0}$. In the sequel we define a natural class of preconditions and show the solvability of CTS problem for it.

We consider tape conditions definable as programs in $\mathbf{L_0}$. One of the most natural ways to do this is as follows. Let S_X be a program in $\mathbf{L_0}$ with one input tape X. A predicate is associated with S_X which is true for those and only those values of tape X where the program S_X stops. Namely this predicate is called *tape condition specified by S_X*.

A tape condition specified by a program could be defined also otherwise. We could say that a tape value X^0 satisfies the condition if and only if X^0 is a correct test for

the program S_X ,i.e., the program S_X when executed on X^0 does not reach a forbidden exit (we allow S_X to be a conditional program). It is not difficult to show by using previous results on program termination that both definitions are uniform, i.e., describe the same class of conditions.

THEOREM 4. *There is an algorithm constructing a finite correct CTS for every program in* $\mathbf{L_0}$ *with preconditions also specified by programs in* $\mathbf{L_0}$.

The proof of the theorem follows from Theorem 3 and Lemma 1.

LEMMA 1. *For every program P in* $\mathbf{L_0}$ *with preconditions specified by programs in* $\mathbf{L_0}$ *there is a conditional program* P^* *in* $\mathbf{L_0}$ *without preconditions such that every correct CTS for* P^* *is a correct CTS for P with its preconditions. There is an algorithm constructing* P^* *from P and its preconditions.*

The idea of the proof of the lemma is simple. Let us assume for the simplicity that P has only one input tape A with a tape condition specified by a program S_A. It is not difficult to see that we can merge the reading of tape A for checking the precondition S_A and the reading of A during the execution of P into a single process. It means that we can build a program that will execute the job of both program P and program S_A between two consecutive reads from tape A. If a statement has a pending exit in program S_A, then the corresponding exit is left pending for "maps" of this statement in P^*. In such a way the program P^* will contain, on the one hand, the maps of all branches of program P and, on the other hand, all the restrictions in the form of forbidden exits imposed by the condition S_A. If P has more than one input tape with corresponding tape conditions, the method just described allows us, first, to insert the check of the first tape condition into P, then that of the second tape and so on. In such a way we can always build the desired program P^*. This completes the proof.

It is easy to see that for programs in $\mathbf{L_0}$ with preconditions in $\mathbf{L_0}$ there also holds an equivalent of Theorem 2 stating the decidability of nontermination.

5 Programs with Other Simple Data Types

So far we have considered only one simple type – integer in the language $\mathbf{L_0}$. The aim of this section is to generalize the previous results to arbitrary simple types with comparison operators defined. There can be a great variety of such types and comparison operators can be defined in a highly different manner in respect to types. For example, let us consider charstring type. The comparison operator "<" can be defined for it according to the lexicographic ordering: $x_1...x_n < y_1...y_m$ if $\exists i_0$ such that $x_1=y_1, \dots, x_{i0-1}=y_{i0-1}$ and $x_{i0}<y_{i0}$, or $n<m$ and $x_1=y_1, \dots, x_n=y_n$ (the ordering adopted in Turbo Pascal). New situations arises for this ordering as there are infinitely many words between some two words x and y and a finite number of words between some other words. The relation "<" can be defined for this type also otherwise: $x_1...x_n < y_1...y_m$ if $n<m$ or $n=m$ and $\exists i_0$ such that $x_1=y_1, \dots, x_{i0-1}<y_{i0-1}$ and $x_{i0}=y_{i0}$, (ordering used by some other Pascal

implementations). In this case there will be only a finite number of words between any two words x, y. This example shows us the variety of possible situations here. To comprise all the cases we use an "axiomatic" approach in this section: the comparison operator will be requested only to satisfy some "axioms" of constructivity. All the types appearing in real programming languages will satisfy these "axioms". On the other hand we will show that these axioms are sufficient to make the CTS construction problem algorithmically solvable. Our aim is to investigate more deeply what is essential and what is not essential for the algorithmic solvability of the CTS construction problem. So,let

$$T_1, T_2, \ldots, T_s$$

be arbitrary simple types with comparison operators $=, \neq, \leq, <$ defined.

Further we assume these operators to be total. As far as the first three operators can be expressed by the last one (using boolean expressions), we assume (without restriction of generalization) only operator "$<$" to be defined a priori for each type.

Let us assume the values of the types considered to be constructive objects, thus algorithms over the domains of these types can be considered. We shall say that operator "$<$" is *constructive* (satisfies the constructiveness "axioms") for the type T if

(1) there is an algorithm $\mathcal{A}$ which, given any x, $y \in T$, determines whether the relation $x < y$ holds;

(2) there is an algorithm $\mathcal{B}$ which, given any x, $y \in T$ such that $x < y$, determines whether there exists $z \in T$ such that $x < z$ and $z < y$ and in the case of existence gives one such z;

(3) there is an algorithm $\mathcal{C}$ which, given any $x \in T$, determines whether there exists $z \in T$ such that $z < x$ and in the case of existence gives one such z;

(4) there is an algorithm $\mathcal{D}$ which, given any $x \in T$, determines whether there exists $z \in T$ such that $x < z$ and in the case of existence gives one such z.

Let us consider the most popular simple types:

– integer with operator "$<$";
– natural with operator "$<$";
– rational with operator "$<$"; (e.g., binary and decimal fixed point data)
– real, as treated by most common programming languages, i.e., floating point data (values are of form $n_1.n_2E + n_3$ with limited precision and limited exponent, in fact, they are rational numbers);
– charstring with operator "$<$" defined in one of the ways considered at the beginning of this section;
– integer subranges and enumerable types (like in Pascal) with operator "$<$".

It is easy to see that all these types are constructive in the abovementioned sense.

So let us assume some constructive types $T_1, \ldots, T_s$ to be fixed. We also assume that every value constant of these types uniquely determines the type to which it belongs.

Now let us consider the following generalization of the language $\mathbf{L_0}$, namely, the language

$$L_0^{T_1,\ldots,T_s}.$$

Programs in $L_0^{T_1,\ldots,T_s}$ like in $\mathbf{L_0}$ will have both internal and external variables. Each internal variable will be of some fixed simple type (to stress that internal variable x is of the type T we sometimes use the denotation x^T). Again tapes will be used as external variables. We suppose that a cell of a tape can contain a value of any simple type. Thus, the value of a tape is an arbitrary finite sequence $(x_1 \ldots, x_n)$ where x_i belongs to some of

the types $T_1,...,T_s$. Just as before be both input and output tapes are used.

Statements in $L_0^{T1,...,Ts}$ are just the same as in L_0. Assignments and comparisons are allowed only between the variables (and constants) of the same type. Some additional comments are necessary for input statement

$$X \to u,$$

where X is an input tape and u is an internal variable of the type T. Let the reading head of the tape X be on the i-th cell at the moment when the statement is executed. If the i-th cell contains a value of the type T, the statement is executed normally, i.e., the value of the i-th cell is assigned to the variable u and the head moves one position right (at the beginning the head was at the first cell). If the i-th cell contains a value of some other type, an error (crash) occurs and the execution of the program is halted.

A natural question arises whether previous theorems can be generalized to programs with arbitrary constructive simple types. We shall consider the analogue of Theorem 1 in some detail.

Further on by a program we understand only a closed program, i.e., a program without pending exits. Let P be such a program in $L_0^{T1,...,Ts}$ with input tapes A,B,...,C. A test $A = A^0$, $B = B^0$,..., $C = C^0$ is said to be *admissible* if the program P does not crash on this test.

A test set T is said to be *a complete test set* for a program P if

(1) it contains only admissible tests,

(2) every branch of the program executable by an admissible test is executed by some test of the set.

THEOREM 5. *Let $T_1,...,T_s$ be fixed constructive types.Then there exists an algorithm constructing complete test set for every program in $L_0^{T1,...,Ts}$.*

The proof of the theorem will be similar to that of Theorem 1, only some lemmas will be more complicated.

Let T be a fixed constructive type with its corresponding algorithms *A, B, C, D*. We need the generalization $x < (r) < y$ of the inequality $x < y$ for $r \in \{0,1,2,\dots\}$:

$x < (0) < y$ means $x \le y$ (i.e., $\neg(y > x)$),

$x < (1) < y$ means $x < y$,

$x < (r) < y$ where $r \ge 2$ means that there are elements $e_1, e_2,...,e_{r-1}$ of the type T such that $x < e_1 < e_2 < ...< e_{r-1} < y$.

By an inequality system N of the type T we understand a system

$$x_1 < (r_1) < y_1$$

$$\dots$$

$$x_p < (r_p) < y_p,$$

where x_i, y_i are variables or constants of the type T.

Such a system of inequalities N (like in Section 2) is represented by a graph G_N : the vertices of the graph are labeled by variables and constants of the system N and an edge of weight r is drawn from vertex y to vertex x if there is an inequality $x < (r) < y$ in the system N. Vertex x is called constant vertex, if it corresponds to a constant in the system N and variable vertex, if it coresponds to a variable. Variable vertices with no edges issuing are called minimal ones. Variable vertices with no incoming edges are called maximal ones. Let us consider a path in the graph G_N. By weight of a path we understand, just as before, the sum of the weights of its edges.

Let us introduce some more notations. Let $x,y \in T$. Let us denote

$M(x,y)=\{ z | z \in T \ \& \ z<x \ \& \ y<z \}$,

$M(*,x)=\{ z | z \in T \ \& \ x<z \}$,

$M(y,*)=\{ z | z \in T \ \& \ z<y \}$.

The cardinality of the set M is denoted by $|M|$ (it can also be infinity).

Let us define

$$x-y=\begin{cases} 0 \text{ , if } x=y \\ |M(x,y)|+1 \text{ if } y<x \\ -|M(y,x)|-1 \text{ if } x<y \end{cases}$$

The "–" operator just introduced coincides with the conventional minus operator in the case when T is the integer type.

LEMMA 1. *An inequality system N of a type T has a solution if and only if its graph G_N has the following properties:*

(1) the weight of every cyclic path is equal to 0,

(2) the weight of every path leading from a constant vertex c_1 to other constant vertex c_2 does not exceed c_1-c_2,

(3) if the type T has the smallest value ω, then the weight of every path leading from a constant vertex c to a minimal variable vertex does not exceed c- ω ,

(4) if the type T has the largest value Ω, then the weight of every path leading from a maximal variable vertex to a constant vertex does not exceed Ω–c,

(5) if the type T has both the smallest value ω and the largest value Ω, then the weight of every path leading from a maximal vertex x to a minimal vertex y does not exceed Ω–ω .

Before we proceed to the proof of the lemma let us remark that the beforementioned algorithms *A, B, C, D* do not yield a constructive method to check the lemma conditions. Therefore, up to now the lemma has only qualitative meaning.

Now let us begin the proof.

The necessity of lemma conditions is obvious.

Let us prove the sufficiency, we assume lemma conditions (1) - (5) to be true. We search the solution by induction. On each step of the induction we assign a constant value of the type T to a variable vertex of G_N. We assign the constant values (i.e., replace the variable vertices by constant ones) so that the validity of lemma conditions is preserved.

To implement this idea we have to make some preparations. At first let p(x,y) denote the maximal weight of the paths leading from vertex x to vertex y.

1) Let us consider all pairs of constant vertices (c_1,c_2) where there is a path from c_1 to c_2. The second condition of lemma implies $c_2 \leq c_1$. For every pair of vertices, where $p(c_1,c_2)$ -1 ≥ 1, we construct elements $e_1, e_2, \ldots, e_{p(c1,c2)-1}$, such that $c_2<e_1<e_2<\ldots<e_{p(c1,c2)-1}<c_1$, using algorithms *A* and *B*. The existence of such elements is provided by the second condition of the lemma.

2) Let us consider all pairs (c,x) where c is a constant vertex, x is a minimal variable vertex and there is a path from c to x. For every pair of vertices, where $p(c,x) \geq 1$, we construct elements $e'_1, e'_2, \ldots, e'_{p(c,x)}$, such that $e'_1<e'_2<\ldots<e'_{p(c,x)}<c$, using algorithms *A, B* and *C*.

3) Let us consider all pairs (x,c) where c is a constant vertex, x is a maximal variable vertex and there is a path from x to c. For every pair of vertices, where $p(x,c) \geq 1$, we construct elements $e''_1, e''_2, \ldots, e''_{p(x,c)}$, such that $c < e''_1 < e''_2 < \ldots < e''_{p(x,c)}$, using algorithms *A*, *B* and *D*. The existence of such elements is provided by the fourth condition of the lemma.

4) Let us consider all pairs (x, y) where x is a maximal variable vertice, y is a minimal variable vertice and there is a path from x to y. For every pair of vertices we construct elements $e'''_1, e'''_2, \ldots, e'''_{p(x,y)+1}$, such that $e'''_1 < e'''_2 < \ldots < e'''_{p(x,y)+1}$, using algorithms *A*, *B*, *C*,*D* (and assuming that we know at least one element of each type). The existence of such elements is provided by the fifth condition of the lemma.

Now let us consider all the beforementioned elements e, e', e'', e''' together with constants of the inequality system. By means of the algorihtm *A* we sort them in ascending order:

$$a_1 < a_2 < a_3 < \ldots < a_u \qquad (*)$$

The elements corresponding to constants of the inequality system are called constant elements, the other ones – auxiliary elements.

We begin to solve the inequality system N by assigning the value a_1 to all minimal variables and the value a_m to all maximal variables. This will introduce new constants in the inequality system N and its graph G_N. It is not difficult to ascertain that the introduction of such constants does not affect the truth of the lemma conditions (1) – (5). In the sequel we have, in fact, to deal only with conditions (1) and (2), the conditions (3) – (5) are used no more.

At first let us deal with vertices which belong to a cyclic path (its weight is 0 by the condition (1)). If there is a constant among them, assign the constant to all variable vertices. If all vertices of the path are variable ones, then select one variable as a representative of the path (for other variables must have the same value) and replace the cyclic path by this variable.

Let us consider a vertex z in the graph G with no constant value assigned to it in the previous steps. Let us take all constant vertices (including the ones introduced in the previous induction steps) with the paths leading from the vertices to z. We make an inductive assumption that all the values of these vertices are within the sequence (*). Hence we suppose these vertices to form a subsequence

$$a_{i1}, a_{i2}, \ldots, a_{im}.$$

of the sequence (*), i.e., every such vertex has a corresponding ordinal number i in the sequence (*). The maximal weights of paths leading from these vertices to z are denoted by $l_1, l_2, \ldots, l_m$ respectively. Let us consider the following elements of the sequence (*)

$$a_{i1-l1}, a_{i2-l2}, \ldots, a_{im-lm}$$

Let us denote by a_h the least of these elements (i.e., the element positioned leftmost in the sequence (*)).

Further we take all constant vertices (including the ones introduced in the previous induction steps) which have paths leading from z to them. Again we make an inductive assumption that values of these vertices form a subsequence

$$a_{j1}, a_{j2}, \ldots, a_{jn}$$

of the sequence (*). Let us denote the maximal weights of paths leading from z to these vertices by $r_1, r_2, \ldots, r_n$, and consider the following elements of (*)

$$a_{j1+r1}, a_{j2+r2}, \ldots, a_{jn+rn}.$$

Let a_g be the largest of these elements.

It follows from the second condition of the lemma that

$a_g < a_h$.

Indeed, if it were not so, it is easy to deduce that constants a_g and a_h would violate the second condition of lemma.

Let us choose any element u_i of the subsequence

$a_g, a_{g+1}, \dots, a_h$

of the sequence (*) as the value z. So the vertex z becomes a constant vertex in the graph G_N. It is easy to observe that lemma conditions are preserved. Moreover, the set of values of constant vertices will not exceed the sequence (*).

Thus we continue the process until all variable vertices in G_N are replaced by constants. These constants form the solution of the system N.

This proves the sufficiency of lemma conditions (and also lemma).

The proof of sufficiency yields us an algorithm for solving inequality systems. Let us express this result as a separate lemma.

LEMMA 2. *For each constructive type T there is an algorithm which, given any inequality system N of the type T,*
(1) finds a solution if such exists,
(2) produces special indication if there is no solution.

Let us make some remarks to the proof of the second assertion of the lemma. If we consider the abovementioned algorihm for solving inequality systems more in detail, we can see that, in case Lemma 1 conditions fail, the algorithm certainly is aborted, i.e., either there are not enough elements $a_1, a_2, \dots, a_u$ or inequality $a_g < a_h$ fails. The aborting definitely occurs after a finite number of steps. It means the algorithm can always "catch" the nonexistence of solution. So it is possible to overcome the nonconstructiveness of Lemma 1 conditions.

Now let P be a program in $L_0^{T1,\dots,Ta}$ and $\alpha = (k_1, \dots, k_r)$ be an initial path in this program. We define the system of inequalities corresponding to path α just as in the proof of Theorem 1. The only difference is that every occurrence of internal variable in the system will have its type ascribed, e.g., t_7^{T3} , u_0^{T1}, etc. The inequality system also contains variables X_i where X is an input tape. These variables occur only in equalities $X_i = t_j^{Te}$ where the instance of the internal variable t_j^{Te} has already the type T_e ascribed to. Relying on this equality we ascribe the same type T_e also to the variable X_i:

X_i^{Te}.

As assignments and comparisons are allowed only for variables of the same type, the inequality system $N(\alpha)$ splits into independent inequality systems according to types:

$N(\alpha) = \{N^{T1}(\alpha), \dots, N^{Ta}(\alpha)\}$.

Obviously there holds

LEMMA 3. *A path α is feasible iff for each of the types T_e the corresponding inequality system $N^{Te}(\alpha)$ has a solution. Any solution of the systems $N^{T1}(\alpha), \dots, N^{Ta}(\alpha)$ with respect to cell variables of input tapes yields a test executing the path α.*

Further we consider each of the inequality systems $N^{T1}(\alpha),\dots,N^{Ta}(\alpha)$ separately. Let $N^T(\alpha)$ be one of these inequality systems. Our aim is to define the T–state $S^T(\alpha)$ for a program after the execution of path α, i.e., part of the program state $S(\alpha)$ referring to the type T. The complete state $S(\alpha)$ is defined as $\{S^{T1}(\alpha),\dots,S^{Ta}(\alpha)\}$.

The idea for the definition of $S^T(\alpha)$ is similar to that used in the proof of Theorem 1, namely, we take the inequality systems $N^T(\alpha)$ and exclude inactive variables. However, a new problem arises: how to choose the constant c_0 used to delimit the weights of edges (see the definition of the exclusion of variable y in the proof of Theorem 1). Let us proceed as follows. Let us consider all constants of the type T in the program P, as well as the smallest and largest values of the type T, if there are such. Sort all these constants in ascending order

$$c_1, c_2, \dots, c_m.$$

Let us consider differences $c_j - c_i$. These differences can be infinite for some pairs and finite for some others. Let us consider all pairs (c_i, c_j) where the difference is finite. Let us denote the largest of the differences by C_0^T.

Now let us define the exclusion of inactive variables from the inequality system $N^T(\alpha)$ just as before, with just defined constant C_0^T playing the role of c_0. Let us recall that the constant C_0^T is used to delimit the weight of edge: if there is a weight $r > C_0^T + 1$, it is replaced by $C_0^T + 1$. It is not difficult to ascertain that after every exclusion of the variable the following assertion holds for the obtained inequality system $N'^T(\alpha)$: $N'^T(\alpha)$ satisfies the conditions of the existence of a solution from Lemma 1 iff $N^T(\alpha)$ satisfies these conditions. By the way, let us note the following easily provable proposition. Had we used some smaller constant instead of C_0^T in variable exclusion, only the following assertion would hold instead of the previous: if $N^T(\alpha)$ satisfies the conditions of solution existence from Lemma 1, then also $N'^T(\alpha)$ satisfies the conditions (but not vice versa).

Now let us define the state $S^T(\alpha)$ to be the inequality system obtained from $N^T(\alpha)$ by excluding all inactive variables. It follows from the above mentioned that an analogue of Lemma 3 from Theorem 1 holds for state $S^T(\alpha)$.

However, state $S^T(\alpha)$ cannot be used directly. The matter is that the constant C_0^T, upon which the construction of state $S^T(\alpha)$ relied, cannot be effectively found for every constructive type T. Therefore we do as follows. For every natural constant c we consider the state $S^{T,c}(\alpha)$, the definition of which differs from the definition of $S^T(\alpha)$ only in the point that constant c is used instead of C_0^T. By the way, if $c = C_0^T$, then $S^T(\alpha) = S^{T,c}(\alpha)$. From the abovementioned there follows

LEMMA 4. *For every constant $c \in N$ the existence of a solution for system $N^T(\alpha)$ implies the existence of a solution for system $S^{T,c}(\alpha)$ (i.e., the consistency of the state $S^{T,c}(\alpha)$). If $c \geq C_0^T$, then also the existence of a solution for the system $S^{T,c}(\alpha)$ implies the existence of a solution for the system $N^T(\alpha)$.*

It is not difficult to see that for every $c \in N$ an analogue of Lemma 4 from Theorem 1 holds for the state $S^{T,c}$.

LEMMA 5. *Let σ be a state of the type $S^{T,c}$ for the program P, α a path in the program and β a continuation of the path α. Then the equality holds:*

$$S^{T,c}(\sigma,\alpha+\beta) = S^{T,c}(S^{T,c}(\sigma,\alpha),\beta).$$

Now let us define for an arbitrary tuple of natural numbers $(c_1,...,c_a)$ general state

$$S^{c1,...,ca}(\alpha) = \{S^{T1,c1}(\alpha),..., S^{Ta,ca}(\alpha)\}.$$

Let us emphasize that, given constants $(c_1,...,c_a)$, the state $S^{c1,...,ca}(\alpha)$ can be effectively constructed for an arbitrary path α. We also emphasize that the number of possible states for program P is finite for a fixed tuple $(c_1,...,c_a)$.

Thus for every tuple of naturals $(c_1,...,c_a)$ we can build for a program P, using states $S^{c1,...,ca}(\alpha)$, its reachability graph denoted by $G^{c1,...,ca}(\alpha)$. Let us consider the properties of this graph.

From Lemma 4 and other previous lemmas there follows an analogue of Lemma 6 from Theorem 1:

LEMMA 6. *For every tuple of naturals $(c_1,...,c_a)$ the feasibility of an initial path α in program P implies the existence of an initial path γ in the reachability graph $G^{c1,...,ca}(\alpha)$ whose projection is α. If conditions*

$$c_1 \geq C_0^{T1},..., c_a \geq C_0^{Ta}$$

hold for the tuple $(c_1,...,c_a)$, then the existence of an initial path γ in the reachability graph whose projection is α implies the feasibility of path α in program P.

Let us consider the covering U of the reachability graph, namely, the set of paths covering all allowed edges of the graph $G^{c1,...,ca}$. A set of paths pr_U in a program is assoaciated with the set U.

From the previous lemma there follows

COROLLARY. *Every tuple of natural numbers $(c_1,...,c_a)$ has a property: if U is a covering of reachability graph $G^{c1,...,ca}$ for program P, then pr_U contains all feasible branches of the program. If in addition*

$$c_1 \geq C_0^{T1},..., c_a \geq C_0^{Ta}, \qquad (**)$$

then all paths in pr_U are also feasible.

These lemmas show that, if we knew the constants C_0^{T1} ,..., C_0^{Ta} for the given program, we could, using Lemma 2, construct CTS for the program just the same way as in the case of Theorem 1. However, the algorithms ***A, B, C, D*** used in the definition of type constructivity do not yield a method to find these constants. Therefore much more complex actions should be performed as in the case of Theorem 1.

Initially as $c_1,...,c_a$, we choose any natural numbers, e.g., $c_1 = 0,..., c_a = 0$. Using these numbers we construct the reachability graph $G^{c1,...,ca}$ and its covering U consisting of finite paths. Just as before we consider pr_U and for each path $\alpha \in$ pr_U construct the inequality system

$$N(\alpha) = \{N^{T1}(\alpha),...,N^{Ta}(\alpha)\}.$$

Then we try to solve these inequality systems using the algorithm from Lemma 2. If the algorithm yields solutions for all paths α, then, as it is implied by the first assertion of the Corollary, these solutions will form CTS. Now let us assume that the algorithm aborts on some inequality system $N(\alpha)$. It means that there is i such that the algorithm will produce the solution inexistence indication when applied to the inequality system $N^{T_i}(\alpha)$. It follows from the definition of the reachability graph and Lemma 4 that the case is possible only for $c_i < C_0^{T_i}$. This inequality means that there are two constants c_k, c_e in program P such that $c_e - c_k < \infty$ and $c_e - c_k > c_i$. Since we know this thing, now we apply the algorithms $\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}$ for the type T_i to all possible pairs of constants in the program and so in a finite number of steps we can construct $p \geq c_i$ elements between some pair of constants c_k and c_e, besides, by using the algorithm $\mathcal{B}$ we can ascertain that there are no more elements between these constants. In the next iteration step we use the number p+1 as a constant c_i .Thus with every iteration step we approximate constants $c_1,...,c_a$ to the constants $C_0^{T_1}$,..., $C_0^{T_a}$. In such a way we assure that after a finite number of steps constants $c_1,...,c_a$ can be reached such that the covering U of the reachability graph $G^{c_1,...,c_a}$ will have the required property: all $\alpha \in pr_U$ will be feasible, i.e., inequality systems $N(\alpha)$ will have solutions. These solutions will form the desired CTS.

This completes the proof of the Theorem.

Natural question arises whether we can generalize other theorems proven in the previous sections for the language $\mathbf{L_0}$ to programs in $L^{T_1,...,T_a}$ with T_1 ,..., T_a being arbitrary constructive types. Using the techniques elaborated in the previous proofs it is not difficult to obtain a positive answer to this question.

Now we return back to the base language $\mathbf{L_0}$ in the next sections. Methods developed for the language $\mathbf{L_0}$ in many cases can be transferred to wider classes of programs.

6 Programs with Stack

Let us consider a language $\mathbf{L_1}$ where a program has additional internal memory - stack. Formally $\mathbf{L_1}$ is obtained from the base language $\mathbf{L_0}$ by adding the following statements:

t→M (respectively **c→M**). The value of variable t (constant c) is added to the stack. We use the capital M to denote the stack. (Push statement).

M→t. The last element of the stack is assigned to variable t and erased in the stack. The statement has two exits: if the stack is not empty, then the exit '+' is used, otherwise use the exit '-'. In the last case the value of t is not changed. (Pop statement).

<u>**THEOREM 6.**</u> *There is an algorithm for constructing a finite complete test set for every program in* $\mathbf{L_1}$.

The proof is based on the slight modification of the notion of the program state and new lemmas about path replace. Our aim is to construct a reachability graph

containing all feasible branches of the program.

Let us consider the construction of the system of inequalities $N(\alpha)$ corresponding to initial path $\alpha = (k_1, k_2, \ldots, k_r)$ for programs in $\mathbf{L_1}$. The construction is similar to that in Section 2. Additionally the initial system $N(\alpha_0)$ has inequality

$$m_0 = 0,$$

where m_0 is a variable denoting the number of stack elements. Further variables M^i with subscript are used to denote the value of the i-th stack element, and m_k are variables denoting the number of stack elements.

Now let us assume that the system of inequalities $N(\alpha_{i-1})$ is already defined. Let m_s be a variable denoting the number of stack elements after the execution of path α_{i-1}. Let us denote the value of m_s by w. Let u_l be a variable denoting the value of variable u after the execution of path α_{i-1}. Let z be the greatest subscript of all variables M with superscript w and let v be the greatest subscript of all variables M with superscript w+1. If there is no variable with superscript w or w+1, then we assume z=0 or v=0. Then we define $N(\alpha_i)$ as system obtained from $N(\alpha_{i-1})$ by adding the following inequalities:

1) If $k_i = (u \to M)$, then equalities

$$m_{s+1} = w+1 \ , \ M^{w+1}_{v+1} = u_l$$

are added. In this case new variables m_{s+1} and M^{w+1}_{v+1} are introduced. The value of variable m_{s+1} is equal to the number of stack elements.

2) If $k_i = (M \to u+)$ and $m_s > 0$, then equalities

$$m_{s+1} = w-1 \ , \ u_{l+1} = M^{w}_{z}$$

are added. New variables u_{l+1} and m_{s+1} are introduced. If $m_s = 0$, then inequalities

$$u_{l+1} < 0 \ , \ u_{l+1} > 0$$

are added to obtain contradictory inequality system.

3) If $k_i = (M \to u-)$ and if $m_s = 0$, then no inequality is added.
If $m_s > 0$, then inequalities

$$u_{l+1} < 0 \ , \ u_{l+1} > 0$$

are added to obtain contradictory inequality system.

4) If $k_i \in \mathbf{L_0}$, then we proceed the same way as defined for the language $\mathbf{L_0}$.

Let us give an example. For $\alpha = (1: A \to u+, 2: u \to M, 3: M \to t+)$ we have ineguality system

$$N(\alpha) = \begin{cases} u_0 = 0 \\ t_0 = 0 \\ m_0 = 0 \\ u_1 = A_1 \\ m_1 = 1 \\ M^1_1 = u_1 \\ m_2 = 0 \\ t_1 = M^1_1 \end{cases}$$

also in our case. So initial path α is feasible iff system $N(\alpha)$ has a solution. Now let us define state inequality system $S(\alpha)$ for our case. Internal variables with maximal subscripts and variables denoting input files are called active variables. Previous rules of variable exclusion will be used also for all variables m_s and M_j^i. Let us exclude all of them and all other inactive variables. We obtain inequality system containing only active variables and constants. The resulting system is also denoted by $S(\alpha)$ and called a *program state* after execution of path α. Easy to see that Lemma 3 from Section 2 is valid also in this case.

Let us consider a path α with the following property: the number of stack elements on path α is equal to or greater than the initial, after the execution of path α it is equal to the initial. Such path is called a *normal path.*

Let α be a normal path (there is no requirement for α to be initial) and σ be an arbitrary program state. Then we define $N(\sigma,\alpha)$ the same way as for the language $\mathbf{L_0}$. $N(\sigma,\alpha_0)$ is the same initial inequality system σ with only zero subscripts added to initial variables. The equality $m_0=0$ is also added to describe the initial status of stack. Further $N(\sigma,\alpha_i)$ is defined from $N(\sigma,\alpha_{i-1})$ and statement k_i just as before. Let us exclude inactive variables, except initial variables from $N(\sigma,\alpha)$. We obtain inequality system containing constants and internal variables with zero subscript and perhaps internal variables with another subscript. Let us replace second type subscripts of all variables by one. The reduced system is denoted by $E(\sigma,\alpha)$ and called a *path effect.*

LEMMA 1. *Let initial path α have two normal continuations β and γ with the same last statement. Let us denote $S(\alpha)$ by σ. If $E(\sigma,\beta)=E(\sigma,\gamma)$, then path $\alpha+\beta$ and path $\alpha+\gamma$ have the same feasible continuations.*

Let path δ be a continuation of path $\alpha+\beta$ or path $\alpha+\gamma$. Let us consider systems $N(\alpha+\beta+\delta)$ and $N(\alpha+\gamma+\delta)$. All variables of systems which are created on path β or γ, except those which are active at the beginning of the path δ, have no inequalities with the variables created on path δ. If we exclude all inactive variables created on path β or γ, we obtain path effect $E(\sigma,\beta)$ and path effect $E(\sigma,\gamma)$. Further we have equivalent systems of inequalities which may differ only by subscripts of the variables created on path δ. There follows the proof of Lemma.

LEMMA 2. *Let initial path α have the continuation $\beta+\gamma+\delta$ where $\beta+\gamma+\delta$ is normal path and γ also is normal path. Let states $S(\alpha)$ and $S(\alpha+\beta)$ are equal. Let path β and path γ have the same first statement and let path γ and path δ have the same last statement. If $E(S(\alpha),\beta+\gamma+\delta)=E(S(\alpha+\beta),\gamma)$, then path $\alpha+\gamma$ and path $\alpha+\beta+\gamma+\delta$ have the same feasible continuations.*

It follows from lemma condition that $E(S(\alpha),\gamma)=E(S(\alpha+\beta),\gamma)$. Then $E(S(\alpha),\gamma) = E(S(\alpha),\beta+\gamma+\delta)$. Now according to Lemma 1 path $\alpha+\beta$ and path $\alpha+\beta+\gamma+\delta$ have the same feasible continuations.This proves the lemma.

The number of the pairs (n_i,S_i), where n_i is the statement label and S_i is the program state, can be estimated by constant R_1 effectively evaluated from the given program. Also the number of quadruples (n_i, S_i, E_i, k_i), where n_i,k_i are statements labels, S_i is a state and E_i is path effect, can be estimated by constant R_2 effectively evaluated from the given program.

LEMMA 3. *For any feasible branch δ there exists a path α such that path $\alpha+\delta$ is feasible and the number of stack elements is less than R_1+R_2 on the path α.*

Let us denote by β feasible initial path to branch δ. Let us assume that the number of stack elements after the execution of β is more than R_1. Then we consider stack elements pushed on path β and not popped on path β. Let us denote by β_{k^i} the initial part of path β to the statement when the i-th abovementioned stack element is pushed and by β'_{k^i} the continuation. If we find β_{k^i} and β_{k^j}, $k^i<k^j$, where $S(\beta_{k^i})=S(\beta_{k^j})$ and first statements of β_{k^i} and β_{k^j} are the same, then path $\beta_{k^i} + \beta'_{k^j}$ is feasible. So we find feasible path β' to the branch δ where the number of stack elements after executing β' is less than R_1.

Let us denote by α the initial part of path β until maximal stack length of path β is reached and by ε the continuation of α. Let us consider the stack elements pushed on path α and popped on path ε. Let us denote by α_{k^i} the initial part of path α to statement, when the i-th stack element is pushed, and by ε_{k^i} continuation to statement, when this element is popped, and by γ_{k^i} continuation to branch δ. Every path ε_{k^i} is normal path and we consider $E(S(\alpha_{k^i}),\varepsilon_{k^i})$. If we find that path ε_{k^i} and path ε_{k^j} satisfy conditions of Lemma 2, then path $\alpha_{k^i}+\varepsilon_{k^j}+\gamma_{k^j}+\delta$ is also feasible. So we find path α with no more than R_1+R_2 stack elements used.

Before we start the construction of reachability graph we must extend the notion of the program state. We must include in the state inequality system all variables denoting the values of stack elements. So additionally the state inequality system has active variables M^1, M^2 ,...,M^t where M^i denote the value of the i-th stack element.The number of active variables denoting stack elements is equal to the value of variable m_p where m_p is variable with maximal subscript. Let us denote by $F(\alpha)$ the extended program state after the execution of initial path α. Let us denote by $F(\omega,\alpha)$ the extended program state after the execution of the path α from the extended state ω.

LEMMA 4. *Let ω be an extended state, α a path and β a continuation of path α. Then $F(\omega,\alpha+\beta)=F(F(\omega,\alpha),\beta)$.*

We can notice that inequalities generated on path β do not contain inactive variables of path α. So while constructing $F(\omega,\alpha+\beta)$ from $N(\omega,\alpha+\beta)$ we at first can exclude inactive variables generated by path α. We obtain $F(\omega,\alpha)$ as intermediate result, and the construction of $F(F(\omega,\alpha),\beta)$ from $N(F(\omega,\alpha),\beta)$ starts on equivalent inequality system. This proves the lemma.

Now we can start the construction of the reachability graph. In our case vertices of the graph are labeled by pairs (n,F), where n is a statement label and F is extended state. To construct the reachability graph we use the same algorithm as for programs in language $\mathbf{L_0}$ with one additional rule: if the number of stack elements exceeds R_1+R_2 in the state of the new vertex, then we do not construct edges from this vertex.

From the construction of the reachability graph and Lemma 3 follow

LEMMA 5. *Every initial path in the reachability graph is feasible.*

LEMMA 6. *A branch β in the program is feasible iff there is an initial path α in its reachability graph whose projection contains β.*

Complete test set is constructed from the reachability graph the same way as in case of the language $\mathbf{L_0}$. This completes the proof of Theorem 6.

7 Programs with Direct Access

Let us extend the language $\mathbf{L_0}$ by adding a new statement

RESET(X)

where X is input tape. The statement returns the input head of tape X to the beginning of the tape. By using this statement we can have the repeated reading of input tape. Let us denote the new language by $\mathbf{L_2}$.

THEOREM 7. *There exists no algorithm for constructing a finite complete test set for every program in $\mathbf{L_2}$.*

A subclass of programs in $\mathbf{L_2}$ with two input tapes with one usage of RESET for each of them is sufficient for non-existence. We consider two-tape automata by Rabin and Scott [13]. These automata may be represented by programs in base language $\mathbf{L_0}$ with two input tapes. Let us denote by $L_A \cap L_B$ the intersection of languages L_A and L_B represented by two-tape automata A and B. The problem of determination of $L_A \cap L_B$ emptiness is known to be undecidable [13]. We shall consider tapes of automata to be two input tapes of a program in $\mathbf{L_2}$. It is easy to construct a program P_{AB} using RESET statement only once for each of the tapes where STOP statement is accessible iff $L_A \cap L_B \neq 0$. Hence it follows that the emptiness of $L_A \cap L_B$ can be decided by means of a complete test set.

The previous theorem indicates that the unsolvability of CTS construction problem tends to appear readily if multiple reading of input tapes is allowed. Nevertheless it is possible to select the natural program classes with direct access by addressing to tape cells and retain the CTS problem solvability.

Further we consider a certain class of the type. In this case input and output tapes are not divided. We use both access methods for every tape. For this purpose the tape cells are addressed by numbers 1,2,3,..., and additionally to internal variables u,v,...,t we introduce a finite number of internal address variables which store tape-cell addresses. Every tape has its own address variables. We use capital letter to denote the tape and a corresponding small letter with superscript to denote the address variable. The address variables of tape A are denoted by $a^1, a^2, a^3, \ldots, a^k$, those of tape B by $b^1, b^2, b^3, \ldots, b^m$, etc. We say that a tape contains the sequence of integers $n_1, n_2, \ldots, n_r$ if integers $n_1, n_2, \ldots, n_r$ are written on the tape beginning from the first cell. As we need to modify the statements of the language $\mathbf{L_0}$ we will repeat the definition of all statements. Let A be an arbitrary tape. Let u,t be arbitrary internal variables and a^i, a^j arbitrary address variables of tape A. A program is constructed using the following statements:

1. **START.** The first statement of the program. This statement transfers heads of all tapes to the beginning and sets values of all internal and address variables to the initial value 0. A program has exactly one START statement.

2. **A → u**. The value of the scanned cell of tape A is assigned to variable u. The statement has two exits: exit "+", when the scanned cell contains a number, and exit "-", when the scanned cell is empty.This statement does not move the head on tape A. (Input statement).

3. **u → A**. The value of variable u is assigned to the scanned cell of tape A. (Output statement).

4. **NEXT(A)**. The head moves right to the next cell of tape A. (Shift statement).

5. **u → t** (respectively **c→t**). The value of variable u (constant c) is assigned to variable t. (Assignment statement).

6. **u < t** (respectively **c<t, u<c**). The statement has two exits: if the value of u (respectively c) is less than the value of t (respectively c), then exit "+" is used, otherwise use exit "-". (Comparison statement).

7. **ADR(A) →** $\mathbf{a^i}$. The address of the scanned cell on tape A is assigned to the address variable a^i. (Address input statement).

8. $\mathbf{a^i \Rightarrow u}$. The value of tape A cell whose address is equal to the value of variable a^i is assigned to variable u. The statement has two exits: exit "+", when the tape A cell contains an integer, and exit "-", when it is empty. (Direct access input statement).

9. $\mathbf{u \Rightarrow a^i}$. The value of variable u is assigned to tape A cell whose address is equal to the value of the address variable a^i. (Direct access output statement).

10. $\mathbf{a^i \rightarrow a^j}$. The value of address variable a^i is assigned to address variable a^j Only address variables of the same tape are allowed in the statement. (Address assignment statement).

11. **STOP.**

Let us denote the language obtained in such a way by $\mathbf{L_3}$

Programs in the language $\mathbf{L_0}$ differ by the following constraints:

1) there is no special statement START, i.e., the exits of other statements cannot lead to START;

2) every input or output statement is directly followed by the shift statement of the corresponding tape;

3) all tapes are divided into input tapes and output tapes.

Now all these constraints are dropped.

The new statements dealing with addresses are used for all tapes. Note, however, that the address assignment statement can be applied only to address variables associated with the same tape. Removing this restriction leads to unsolvability of the problem of construction of complete test set. The same result is also obtained if we allow a comparison statement for address variables. In reality, these restrictions usually hold. The new language can be used to code the bubblesort algorithm. Fig.13 gives an example of such a program.

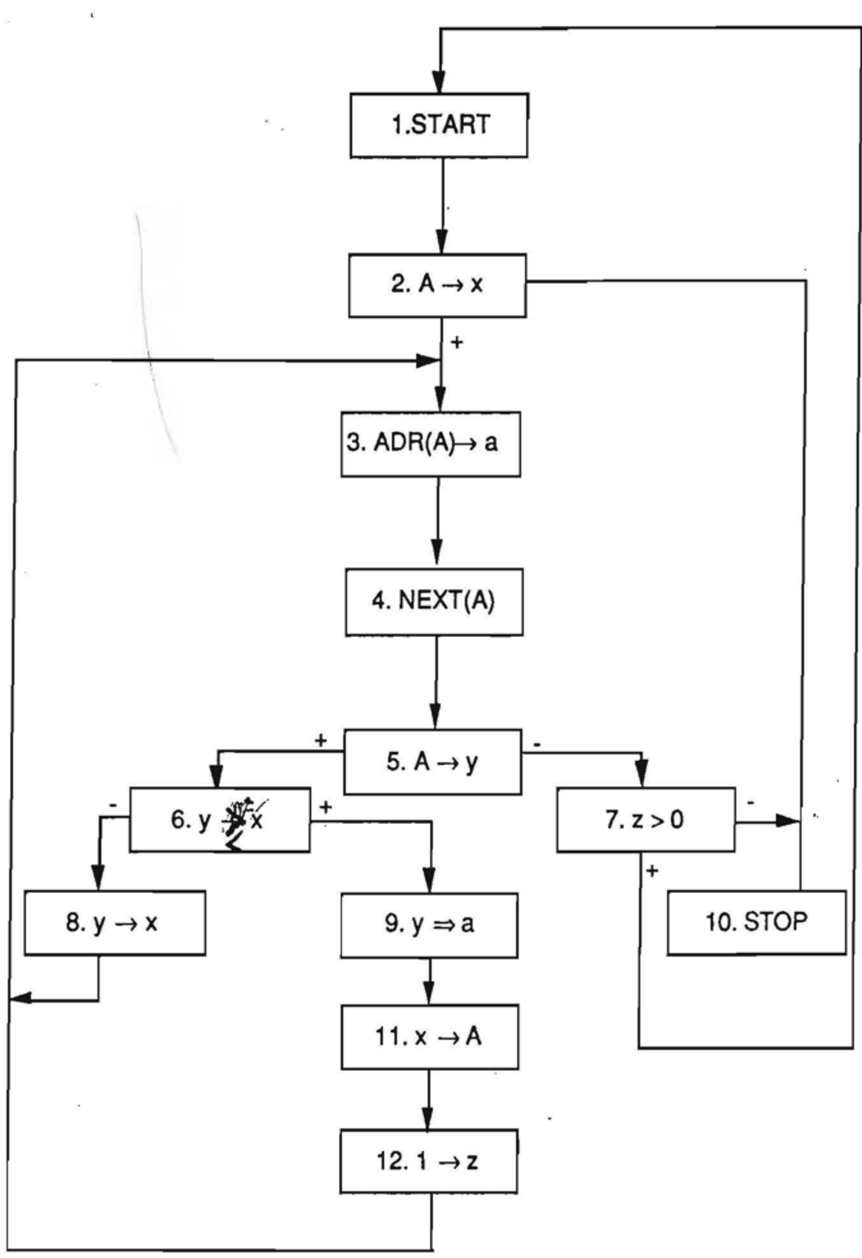


Fig. 13

THEOREM 8. *There exists an algorithm for construction a complete test set for any program in the language* L_3.

At first we can notice that we must consider only those paths where statement START is executed only once. The content of tapes after the execution of START statement is considered as initial.

Further we use a new concept for program state.

Let us interrupt the program execution on fixed tape values. Now all variables have some fixed values. Let c_{min} be the smallest constant and c_{max} be the largest constant of the program. Let us note on the number line all integer values from c_{min} to c_{max} and all variables according to their values. Such ordering of variables and constants is called a configuration. For example, if c_{min} =-1, c_{max} =2, u=-5, v=-5, t=1, z=7, then we obtain

```
u  -1  0  1  2  z
v         t
```

For values less than c_{min} or more than c_{max} only relations between them are important. So, if u=-6 and v=-6 or z=10, then we obtain the same configuration. Configuration is considered to be a special type inequality system. Further not only internal variables and constants are noted in the configuration but also the values of scanned cells and the values of cells whose addresses are equal to the values of address variables. Tape A scanned cell value is denoted by A↓ and the value of cell whose address is equal to the address variable a^i by a^i↓.

Let us involve the new concept of the program state. The state is triple [d,Q,R], where

D is the state of all tapes D=(d_A ,d_B ,...,d_Z), where d_X=0, if the tape X is exhausted, otherwise d_X =1,

Q is the set of configurations,

R is the list of sets of address variables. The list contains sets of address variables with address variables of one set containing the same address. Likewise the sets may also contain the address of the scanned cell of every tape. In the list we denote the address of the scanned cell of tape A by **A**, that of tape B by **B**, etc.

Now we can start the construction of the reachability graph.

The initial state S_0 is [D,Q,R] where D=(1,1,...,1), Q contains one configuration K_0 and R is empty. K_0 is a configuration where all variables are located at constant 0. The constant 0 is always noted in a configuration, as it is the initial value of variables.

The first vertex is labeled by the pair (n_0 ,S_0) where S_0 is the initial state and n_0 is the label of the START statement. Then we construct the edge from vertex (n_0 ,S_0) to vertex (n_1 ,S_1), n_1 is the label of the next statement and S_1 is the state [D,Q,R] where set Q contains all configurations that are obtained from the configuration K_0 by adding variables A↓, B↓,..., Z↓. The adding of variables is described further when we consider shift statement.

Further we define construction rules for all other statements. Let us consider an arbitrary vertex (n,S) with S being the state [D,Q,R].

1) If n = (NEXT(A)) and the statement exit leads to statement n_1, then the new vertex is (n_1, S_1). If d_A=1 in the state S, then state S_1 is $[D,Q_1,R_1]$ where set Q_1 contains all configurations which can be obtained from set Q configurations by transferring or deleting variable A↓. R_1 is obtained from R by deleting **A** from all sets. In the case sets with one element appear they are also deleted.

Actually the deleting of variable A↓ from configuration means that the current scanned cell is empty.

Variable transferring means its deleting from the current point and adding to any point of configuration.

For example, if Q contains a configuration

```
0  1  2  t
   u
```

and A↓ must be added, then 7 possible configurations are obtained:

```
   0 u 2 t      0 u 2 t      0 u 2 t      0 u 2 t
A↓              A↓             A↓             A↓

0 u 2    t      0 u 2 t      0 u 2 t
      A↓             A↓              A↓
```

If d_A=0, then state S_1 is $[D,Q,R_1]$.

2) If n=(A→u) and the exit "-" leads to statement n_1 and exit "+" to statement n_2, then new vertices are constructed the following way. If d_A=0 in state S, then we construct an edge from vertex (n,S) to vertex (n_1,S_1) where S_1 is [D,Q,R]. If d_A=1, then let us denote by Q_1 the subset of Q where configurations contain variable A↓ and by Q_2 the set $Q \backslash Q_1$. If set Q_1 is not empty, then an edge to vertex (n_2,S_2) is constructed. State S_2 is $[D,Q_3,R]$ where configurations of set Q_3 are obtained from configurations of set Q_1 by transferring variable u to the point where variable A↓ is noted. If set Q_2 is not empty, then an edge from vertex (n,S) to vertex (n_1,S_3) is also constructed. State S_3 is $[D',Q_2,R]$ where D' is obtained from D by setting d_A to 0.

3) If n=(u → A) and the exit leads to n_1, then the new vertex is (n_1, S_1). State S_1 is [D,Q',R] where configurations of Q' are obtained from set Q configurations by transferring A↓ to the point where variable u is noted. If list R contains a set T with **A**, then all variables of set T are also transferred to the point where variable u is noted.

4) If n=(u → v) and exit leads to n_1, then the new vertex is (n_1, S_1). State S_1 is [D,Q',R] where configurations of Q' are obtained from configurations of Q by transferring u to the point where variable v is noted.

5) If n=(u < v) and exit "-": leads to statement n_2 and exit "+" to statement n_1, then the new vertices are constructed the following way. Let us denote by Q_1 the subset of configurations from Q where u is noted on the left from v and by Q_2 the set $Q \backslash Q_1$. If Q_1 is

not empty, then an edge to vertex (n_1,S_1) is constructed where S_1 is $[D,Q_1,R]$. If Q_2 is not empty, then an edge to vertex (n_2,S_2) is constructed where S_2 is $[D,Q_2,R]$.

6) If $n=(\text{ ADR(A)} \rightarrow a^i)$ and the exit leads to n_1, then the new vertex is (n_1,S_1). State S_1 is $[D,Q',R']$ where configurations of Q' are obtained from set Q configurations by transferring $a^i\downarrow$ to the point where variable $A\downarrow$ is noted, if $A\downarrow$ is noted in the configuration, or by deleting $a^i\downarrow$, if $A\downarrow$ is not noted in the configuration. List R' is obtained from R by adding set $\{\mathbf{A},a^i\}$, if list R does not contain a set with **A**, or by adding a^i to the set with **A**, otherwise.

7) If $n=(a^i \Rightarrow u)$ and the exit "-" leads to statement n_2 and the exit "+" to statement n_1, then the new vertices are constructed the following way. Let us denote by Q_1 the subset of Q where configurations contain variable $a^i\downarrow$ and by Q_2 the set $Q\backslash Q_1$. If Q_1 is not empty, then an edge to vertex (n_1,S_1) is constructed. S_1 is $[D,Q_1',R]$ where configurations of Q_1' are obtained from set Q_1 configurations by transferring u to the point where variable $a^i\downarrow$ is noted. If Q_2 is not empty, then also an edge to vertex (n_2,S_2) is constructed,where S_2 is $[D,Q_2,R]$.

8) If $n=(u \Rightarrow a^i)$ and the exit leads to n_1, then the new vertex is (n_1,S_1). State S_1 is $[D,Q',R]$ where configurations of Q' are obtained from set Q configurations by transferring $a^i\downarrow$ to the point where variable u is noted. If R contains a set T with a^i, then all variables of set T are also transferred to the point where variable u is noted.

9) If $n=(a^j \rightarrow a^i)$ and the exit leads to n_1, then the new vertex is (n_1,S_1). State S_1 is $[D,Q',R']$ where configurations of Q' are obtained from set Q configurations by transferring $a^i\downarrow$ to the point where variable $a^j\downarrow$ is noted if the configuration contains $a^j\downarrow$. List R' is obtained from R by adding set $\{a^i,a^j\}$ if list R does not contain a^j, or by adding a^i to the set with a^i, if R contains a^j.

If the new vertex already exists, then the edge is joined to the old vertex. When no new edges can be constructed the construction of the reachability graph is finished.

In such a way we receive a lot of paths in the reachability graph. It follows from the construction of the reachability graph that for any feasible path α of the program where START statement is executed only once there exists path α' in the reachability graph whose projection is path α.

LEMMA 1. *Every initial path in the reachability graph is feasible.*

Let us denote by $\alpha=((n_0,S_0),(n_1,S_1), \ldots,(n_r,S_r))$ an arbitrary initial path in the reachability graph. Let us consider the sequence of configurations $K_0,K_1,\ldots, K_r$ where for every $j=0,\ldots,r$ K_j is a configuration from state S_j configuration set and for every $j=0,\ldots,r-1$ the configuration K_{j+1} can be obtained from K_j applying abovedescribed construction rule corresponding to the statement n_j.

Let us denote by $\alpha_i=((n_0,S_0),(n_1,S_1), \ldots,(n_i,S_i))$ the initial part of path α. Let us prove by induction on i that for every path α_i there is a test T_i such that program traverses path α_i on the test T_i and program variables satisfy configuration sequence $K_0,K_1,\ldots, K_i$.

The test T_0 is empty tapes.

If n_i is NEXT(X) and K_{i+1} contains variable $X\downarrow$, then test T_{i+1} is obtained from the test T_i by adding to tape X a cell with value s that locates $X\downarrow$ at the right place in the configuration K_{i+1}. Let us consider possible variable $X\downarrow$ relations with other variables or constants in the configuration K_{i+1}. At first we can notice that every variable except $X\downarrow$ has a fixed value that is received from a tape or is equal to some constant of program. If $X\downarrow$ is located at the point where some variable is noted or the point is in interval $[c_{min}, c_{max}]$, we take the value s equal to the value of the noted variable or constant. If $X\downarrow$ is located at the last point of the configuration, we take as value s the largest value of the configuration increased by one. If $X\downarrow$ is transferred to the first point, we will take as value s the smallest value of the configuration decreased by one. The last case is that $X\downarrow$ is located just between z and v, where z and v denote variables or constants. It is easy to see that only one of them can be equal to constant. If the difference of z and v values is less than 2, we must change the previous values of the tapes of the test T_i. If $v>c_{max}$, then we replace all values of tapes which are equal or greater than the value of v by the same value increased by one. If z is less than c_{min}, then we can decrease by one all the values of tapes equal or less than the value of z. This operation has no influence on the sequence of configurations $K_0, K_1, \ldots, K_i$ and the program traverses on the updated test T_i the same path α_i. But now the difference of z and v values is 2, and we can choose the integer value s satisfying the configuration K_{i+1}.

If K_{i+1} does not contain variable $X\downarrow$ then $T_{i+1}=T_i$.

If n_i is START, then T_{i+1} is obtained by adding a cell to every tape X, such that configuration K_{i+1} contains variable $X\downarrow$.

If n_i is another statement, then $T_{i+1}=T_i$.

This proves the lemma.

Now the usual algorithm constructing CTS from the reachability graph can be used. This completes the proof of Theorem 8.

8 Programs with Counters

At first, it is clear that, if we consider programs with two–way counters and comparisons between a counter and a constant, the problem of CTS construction is algoritmically unsolvable even in the case of two counters. This follows from the well–known result (see, e.g., [12]) that every recursive function can be computed by a special coding on so–called Minsky machine using only two counters Z_1 and Z_2 and statements

$\boxed{z_i + 1 \to z_i}$, $\boxed{z_i - 1 \to z_i}$, + $\boxed{z_i = 0}$ –, $\boxed{\text{STOP}}$.

Therefore, we can hope at best for the solvability of CTS construction problem in the case of one–way counters. However, as the next theorem shows, the algorithmic unsolvability appears quickly also in this case.

So let us denote by $\mathbf{L_4}$ the language obtained from the base language $\mathbf{L_0}$ by adding internal variables of a new type – counters and the following statements for them:

$c \to Z$. The value of constant c is assigned to counter Z.

$Z + 1 \to Z$. Counter Z is incremented by 1.

$Z < t$. The value of counter Z is compared with the value of internal variable t. The statement has two exits: "+" and "-".

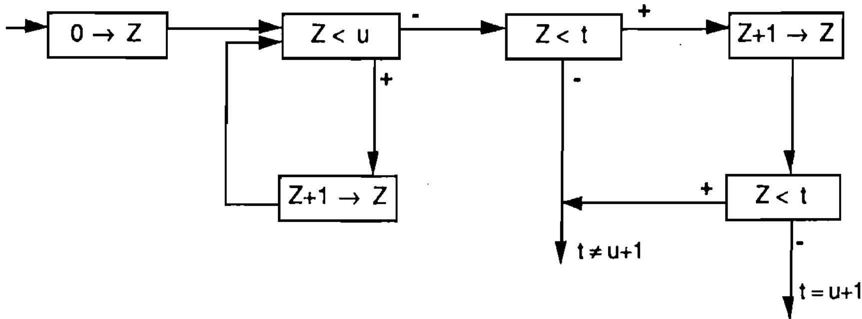
The last statement allows us to compare the value of a counter with the values of an input tape.

__THEOREM 9__. *There exists no algorithm for constructing a complete test system for every program in* L_4. *(The subclass of* L_4 *programs with one input tape and one counter is sufficient for the nonexistence of algorithm).*

The proof of the theorem relies on testing, by means of constructions of language L_4, whether the input tape contains a configuration sequence of some Minsky machine. More detailed it means the following. Let M be a Minsky machine with two counters Z_1 and Z_2. Let us assume the initial value y to be always assigned to the first counter Z_1. Then by a configuration sequence corresponding to the initial value y we understand the following sequence of integers

$$Z_1^0, Z_2^0, \ldots, Z_1^i, Z_2^i, \ldots,$$

where Z_1^0, Z_2^0 are the initial values of counters: $Z_1^0 = y$, $Z_2^0 = 0$, and Z_1^i, Z_2^i are the values of counters after the execution of the i–th step of machine M. For example, if machine M executes statement $Z_1+1 \to Z_1$ on the i–th step, then $Z_1^i = Z_1^{i-1}+1$ and $Z_2^i = Z_2^{i-1}$. If machine M stops on the k–th step (i.e., STOP statement is executed), then configuration sequence is terminated upon Z_1^k, Z_2^k. We shall say in this case that the configuration sequence of machine M is finite for the initial value y. We can ascertain easily that it is possible for any machine M and initial value $y \in N$ to build a program $P_{M,y}$ in L_4, such that $P_{M,y}$ reaches STOP statement only if the configuration sequence of machine M is finite for the initial value y and this sequence is written on the input tape of $P_{M,y}$. In addition the program $P_{M,y}$ uses only one counter Z. The idea of construction of $P_{M,y}$ relies on the fact that it is possible to check whether the relation t=u+1 holds by means of one counter Z (for $u \geq 0$):



Hence it is possible to determine whether the integer sequence written on the input tape is the configuration sequence of machine M for the initial value y. Thus the halting problem for an arbitrary Minsky machine M and initial value y can be reduced to the STOP statement reachability problem for program $P_{M,y}$. Hence follows the algorithmic unsolvability of CTS construction problem for programs in $\mathbf{L_4}$.

Let us denote by $\mathbf{L_5}$ a languge differing from $\mathbf{L_4}$ only in the fact that the counter values can be compared solely with constants. It is easy to see that CTS construction problem is algorithmically solvable for programs in $\mathbf{L_5}$. The same ideas as for the proof of Theorem 1 could be used. The difference is that the counter values, if they lie between minimal and maximal constants of the program, are included in the state.

Further research is connected with finding such restrictions on counters that the solvability of CTS problem is preserved.

A.G.Tadevosjan [14] has considered the following generalization of the language $\mathbf{L_5}$ where together with the beforementioned statements of $\mathbf{L_5}$ the following statement is admitted:

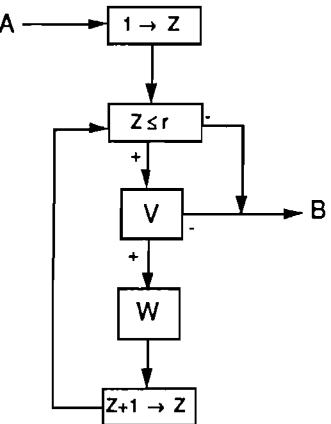
$$t \to Z,$$

where t is an arbitrary internal variable. The CTS construction problem appears to be solvable also in this case.

In practice counters are mainly used for loop organisation. This is done by means of DO statement:

DO Z =1 TO r WHILE V; W; END;

where W – the body of the loop is a program block (by program block we understand part of the program consisting of $\mathbf{L_0}$ statements and, possibly, DO statements and having a single entry and a single exit), V is boolean expression constructed from comparisons of $\mathbf{L_0}$ (e.g., $(t < u)$ & $(5 < t)$), and r – the bound of the loop is an internal variable. DO statement (called also a DO–loop) is interpreted as an abbreviation of the following program block:



It is assumed that counter Z is used in no statements other than the above mentioned ones $Z \leq r$ and $Z + 1 \rightarrow Z$ used for loop organization.

Let us consider the programming language generated by statements of L_0 and DO statement. There is no algorithm for constructing a CTS for every program in this language (a stronger version of Theorem 9). The proof is close to the one used for Theorem 9 except that a slightly different coding of Minsky machine configurations is used. This proof of unsolvability strongly relies on comparing the loop bound r with other internal variables. Now let us exclude this possibility.

We shall not allow the use of the loop bound r in comparisons with other internal variables and in assignments. This means that the loop bound r along with the loop organization statement $Z \leq r$ can be used only in input statement ($X \rightarrow r$), comparisons with constants ($r < c$, $c < r$) and output statements ($r \rightarrow Y$). In practice these restrictions are not essential but they usually hold for real programs. Let us note that several DO–loops can have a common bound r. The programming language generated by the base language statements and the DO statement with the above mentioned restrictions is called L_6.

<u>**THEOREM 10.**</u> *There exists an algorithm for constructing a finite complete test set for every program in* L_6.

A detailed proof of Theorem 10 is rather lengthy, so we shall outline only the main ideas. By a simple state we understand a state in the sense of Theorem 1, i.e., the one obtained by ignoring the statements containing counters and loop bound. Let us consider a DO–loop having no nested DO–loops in it. By entering the DO–loop in a simple state S (at entry point A) and going through all possible values of bound r we can obtain, at the exit of the loop (point B), generally speaking, distinct simple states S_1, $S_2, \ldots, S_n$. Further, for every state S_i there exists the set R_i of the value bound r for which the state S_i is reached at the exit. More precisely, $r' \in R_i$ iff for $r = r'$ there exists a feasible path through the DO–loop beginning at the point A in the state S and reaching the point B in the state S_i .The set R is said to be regular if there exists a regular expression R' in the binary alphabet $\{1, 0\}$ such that for $r \geq 0$

$$r \in R \text{ iff, } \underbrace{11\ldots1}_{r} \in R'$$

and for $r < 0$

$$r \in R \text{ iff, } \underbrace{00\ldots0}_{-r} \in R'$$

The expression R' is said to be a regular representation of the set R. Regular expressions are preferable due to the decidability of the emptiness problem.

<u>**LEMMA 1**</u>. *Set R_i is regular for every i. States $S_1, S_2, \ldots, S_n$ and the corresponding regular representations of sets $R_1, R_2, \ldots, R_n$ can be constructed effectively from the DO–loop and state S.*

Theorem 10 can be proved by Lemma 1 in the simplest case when the program contains only non–nested DO–loops, none of which includes statements involving bounds of other DO–loops. In the general case some generalization of Lemma 1 is necessary.

Let us order the variables used as loop bounds in the program:

$r^1, r^2, \ldots, r^k$

Let us consider a set of strings of the type

$[\, r_a{}^1, r_b{}^1, r_a{}^2, r_b{}^2, \ldots, r_a{}^k, r_b{}^k,]$,

where $r_a{}^j \in \mathbf{N}$, $r_b{}^j \in \mathbf{N} \cup \{*\}$, $\mathbf{N}$ is the set of integers and * a special symbol.

A set of strings is said to be regular if it can be expressed as a finite union of cartesian products of regular set ({*} is considered to be a regular set):

$R_{a,1}{}^1 \times R_{b,1}{}^1 \times \ldots \times R_{a,1}{}^k \times R_{b,1}{}^k \cup \ldots \cup R_{a,m}{}^1 \times R_{b,m}{}^1 \times \ldots \times R_{a,m}{}^k \times R_{b,m}{}^k$,

$R_{a,1}{}^1, R_{b,1}{}^1, \ldots, R_{a,m}{}^k, R_{b,m}{}^k$ - regular sets.

The expression

$R'_{a,1}{}^1 \times R'_{b,1}{}^1 \times \ldots \times R'_{a,1}{}^k \times R'_{b,1}{}^k \cup \ldots \cup R'_{a,m}{}^1 \times R'_{b,m}{}^1 \times \ldots \times R'_{a,m}{}^k \times R'_{b,m}{}^k$,

where

$R'_{a,1}{}^1, R'_{b,1}{}^1, \ldots, R'_{a,m}{}^k \times R'_{b,m}{}^k$

are regular representations of the sets

$R_{a,1}{}^1, R_{b,1}{}^1, \ldots, R_{a,m}{}^k, R_{b,m}{}^k$,

is said to be a regular representation of the corresponding set of strings.

Let a program block with entry C and exit D be given. Let S be a simple state at point C and S_i be a simple state accessible at exit D. Let us denote by U_i the following set of strings:

$[\, r_a{}^1, r_b{}^1, r_a{}^2, r_b{}^2, \ldots, r_a{}^k, r_b{}^k,] \in U_i$

iff for $r^1=r_a{}^1, \ldots, r^k=r_a{}^k$ there exists a feasible path β through the block, beginning at the point C in the state S, and reaching the point D in the state S_i, satisfying the condition: if $r_b{}^j=*$, then the path contains no input statements of the type "tape" $\rightarrow r^j$, if $r_b{}^j$ is a number, then the path contains one or several input statements "tape"$\rightarrow r^j$ and $r_b{}^j$ is a possible value of variable r^j at point D on the path β (j=1, 2, . . . , k).

<u>**LEMMA 2.**</u> *The set of strings U_i is regular for every i. The possible states $S_1, S_2, \ldots, S_n$ at the exit of the block and the corresponding regular representions of the sets $U_1, U_2, \ldots, U_n$ can be constructed effectively from the program block and state S.*

The lemma is proved by induction on the depth of nesting of loops in the block. For depth 1 Lemma 2 is a slight strengthening of Lemma 1.

Now let us consider a block path $\alpha=(k_1, k_2, \ldots, k_m)$. It differs from the usual path in that k_i can be either a $\mathbf{L_0}$ statement or a DO–loop. If k_i is a DO–loop, we fix one of the possible simple states S_i at its exit. An instance of a block path is $\alpha=(X\rightarrow a+,\ (D,S_i),\ a\rightarrow Y)$ where D is some DO–loop. Now let us define the total state $Z(\alpha)$ as a pair $(S(\alpha), W(\alpha))$ where $S(\alpha)$ is a simple state and $W(\alpha)$ is a regular expression describing all possible strings $[r'^1, \ldots, r'^k]$ of numbers acceptable at the end of the path. $W(\alpha)$ can be easily constructed using Lemma 2. It follows from the construction that for a given program the number of distinct total states is finite.

Now using arguments analogous to those used in the proof of Theorem 1 we obtain the proof of Theorem 10.

9 Programs with Real Time Counter

In this section we consider a programming language (let us call it L_7) allowing the simulation of a comparatively wide class of real time systems. At the same time L_7 occurs to have a solvable CTS construction problem. So we have a method for CTS construction for a certain class of real time systems (we show an example how to apply L_7 and its CTS construction algoritm to the analysis of real time systems in the next section).

Programs in L_7 have a finite number of input tapes (output tapes are inessential for CTS construction) and a finite number of internal variables.

The first difference of language L_7 from L_0 is the replacement of integers by rationals in L_7, i.e., cells of input tapes and internal variables have rational values. Thus a test for a program in L_7 will be a fixed assocation of rationals to cells of input tapes.

Let X be an input tape, t,u internal variables and c a rational constant. L_7 have the same statements for these objects as L_0:

1. **X→t**. The statement has two exits just as in L_0; "+", if the current cell is nonempty and "-", otherwise. (Input statement).

2. **u→t** (respectively **c→t**). The value of the variable u (constant c) is assigned to the variable t. (Asignment statement).

3. **t<u**. The statement has two exits "+" and "-", determined by the result of comparison. (Comparison statement).

4. **STOP**.

Besides that every program in L_7 can have one special internal variable z (named *real time counter*) with the following statements permitted:

5. $\mathbf{t \xrightarrow{+} z}$. The statement has two exits "+" and "-". If t>z, then the value of t is assigned to z and the exit "+" is used. If t≤z, then the value of z is not changed and the exit "-" is used. (Positive assignment statement).

The statement can be explained by means of the following block:

```
 ┌──────────┐
 │ t>z ──┬─ -
 │  |+   │ -
 │ t →z  │
 └──┬────┘
    |+
```

6. **z+c→t**. The value of z increased by c is assigned to the variable t. Only nonnegative rational constants c are allowed here. (Activation statement for the variable t).

When modelling real time systems the activation statements will be used to model the activation of the associated timers, the positive assignment statement will correspond to the treatment of input signals (both from outside the system and from timers).

We assume that all internal variables are set to 0 in the beginning.

We can assume without loss of generalization that constants used in the statements of type 2 and 6 are integers (a program in L_7 with arbitrary constants can be transformed to a program with integer constants by changing the scale of number line, easy to see that program logic will not be affected by the scale change).

No principial casualties would appear if rational negative constants were permitted in variable activation statements but there is no real need for it.

Let us note that if we allow normal assignment $t \rightarrow z$ instead of positive asignment, a language with algorithmically unsolvable CTS construction problem would be obtained.

The main result in the section is

THEOREM 11. *There is an algorithm constructing finite CTS for every program in L_7.*

Now let us prove the theorem.

At first for every path α we define the state $S(\alpha)$ corresponding to it, further we use the states to construct the reachability graph (as in the proofs for previous theorems).

Let us call the constants used in assignment statements *basic constants* and the ones in variable activation statements *counter constants* (0 is assumed to be both basic and counter constant by definition). The maximal and minimal basic constants are denoted by c^0_{min} and c^0_{max} , the maximal counter constant is denoted by c^1_{max} (let us remind that we assume all constants to be integers). We define $c_{max}=\max\{c^0_{max}, c^1_{max}\}$.

Let a program P have input tapes A,B,...,C, internal variables $t^1,...,t^k$ and real time counter z. Let us assume the program P to be executing on some test T, i.e., on fixed values of input tapes $A_0, B_0, ..., C_0$. If we suspend the execution at some time moment, we find that the tuple of internal variables $(z,t^1,t^2,...,t^k)$ has a definite numeric value from $\mathbb{Q}^{k+1}$ (here $\mathbb{Q}$ is the set of rational numbers). So at the fixed moment of P execution we may consider as numeric values:

1)all basic constants of the program P,

2)all internal variables t^i and z,

3) $z, z+1, ..., z+c_{max}$ (let us call them further *active points*).

Every one of these values is located somewhere on the number line, thus defining the ordering of basic constants, internal variables and active points. Let us call this ordering an *absolute configuration* (absolute ordering) of P variables at the fixed moment.

Variables t^i and basic constants c whose values at the given moment are within segment $[z, z+c_{max}]$ are said to be *active variables* and *active constants* at this moment. Further the segment $[z, z+c_{max}]$ is called *z-interval.*

For every active variable t^i (active constant c) we define its *relative offset* t^i_Δ (c_Δ respectively) as the difference between t^i (c) and its nearest active point on the left: $t^i_\Delta = (t^i - z) - [t^i - z]$ ($c_\Delta = (c-z) - [c-z]$); here $[x]$ denotes the integer part of x.

The ordering of relative offsets of active variables and constants on number line (with number 0 included) is called the *relative configuration* (relative ordering) of P variables at the given moment.

The pair consisting of absolute and relative configurations of P variables is called *variable configuration* of P at the given moment (while working on the fixed values of input tapes $A_0, B_0, ..., C_0$).

Example. If at the given moment of P execution $z=7.15$, $t^1=2.43$, $t^2=6.86$, $t^3=7.68$, $t^4=9.65$, $t^5=10.15$, $t^6=14.30$ and P has basic constants 1 and 3, $c^1_{max} = 4$, then P has the following variable configuration:

$0 < 1 < t^1 < 3 < t^2 < z < t^3 < z+1 < z+2 < t^4 < t^5 = z+3 < z+4 < t^6$,

$0 = t^5_\Delta < t^4_\Delta < t^3_\Delta$.

The definition of the variable configuration at the given moment utilizes only the values of internal variables at the moment and the values of P constants. Thus, if we know a priori the values of P constants, we can likewise define the P-configuration for every tuple of k+1 rationals $(z, t^1, \ldots, t^k) \in \mathbb{Q}^{k+1}$.

We define the variable configuration set for program P to be the set consisting of P-configurations corresponding to all tuples $(z, t^1, \ldots, t^k) \in \mathbb{Q}^{k+1}$ where $z \geq 0$. It is easy to see that the variable configuration set $\mathcal{C}_P^{Var}$ is finite for every program P in $\mathbf{L}_7$.

Since every variable configuration is actually an inequality system, then, in case a configuration C corresponds to a tuple of variables **t**, we say that **t** *satisfies* C or **t** is one of the solutions of C.

Let k be a statement in the programm P and ε an exit of it. We define the relation $-(k,\varepsilon)\rightarrow \in \mathcal{C}_P^{Var} \times \mathcal{C}_P^{Var}$ for this statement and exit the following way:

1) if $k = (c \rightarrow t^i)$ or $k = (t^j \rightarrow t^i)$ or $k = (z+c \rightarrow t^i)$, then $C_1 -(k,\varepsilon)\rightarrow C_2$ iff the configuration C_2 can be obtained from C_1 by erasing t^i and t^i_Δ (retaining the absolute and relative ordering of all other variables and constants) and then locating them at c and c_Δ, or at t^j and t^j_Δ, or at z+c and 0 respectively (if c_Δ or t^j_Δ is undefined respectively, t^i_Δ also remains undefined);

2) if $k = (X \rightarrow t^i)$ and ε="+", then $C_1 -(k,\varepsilon)\rightarrow C_2$ holds iff C_2 can be obtained from C_1 the following way:

(i) just as before, we erase t^i and t^i_Δ in C_1;

(ii) we locate t^i at an arbitrary place in the absolute ordering of the configuration obtained;

(iii) if t^i in the absolute ordering is placed within z-interval, we locate t^i_Δ in the relative one at any place not contradicting with the place of t^i in the absolute ordering (if $t^i = t^j$ in the absolute ordering, then, certainly, $t^i_\Delta = t^j_\Delta$ in the relative one, if t^i is located so that $z+c \leq t^j < t^i < t^s \leq z+c+1$ ($c < c_{max}$), then $t^j_\Delta < t^i_\Delta < t^s_\Delta$ should hold in the relative ordering);

3) if $k = (X \rightarrow t^i)$ and ε="-", then $C_1 -(k,\varepsilon)\rightarrow C_2$ iff $C_1=C_2$;

4) if $k = (t^i < t^j)$, or $k = (t \xrightarrow{+} z)$ and ε="-", then $C_1 -(k,\varepsilon)\rightarrow C_2$ iff C_1 does not contradict with the exit of the statement and $C_1=C_2$;

5) if $k = (t^i \xrightarrow{+} z)$ and ε="+", then $C_1 -(k,\varepsilon)\rightarrow C_2$ holds iff C_2 can be obtained from C_1 the following way:

(i) If $t \leq z$, then C_2 does not exist.

(ii) If $t > z$, then proceed as follows:

Step 1 (deterministic).

At first we determine the mutual ordering and allocation with respect to active points of the configuration C_2 under construction for the variables and constants which

are on the left (below) $z+c_{max}$ in C_1. To do this we define (calculate) for each variable t^j and basic constant c which are within segment $[\, t^i, z+c_{max}\,]$ in C_1 the value

$$\delta(t^j) = [t^j-t^i] = [t^j-z] - [t^i-z] + \Delta_{ji} \text{ where } \Delta_{ji} = 0, \text{ if } t^i_\Delta \le t^j_\Delta, \text{ and } \Delta_{ji} = -1, \text{ if } t^j_\Delta < t^i_\Delta,$$

here $[t^j - z]$ and $[t^i - z]$ can be inferred from the allocation of t^j and t^i with respect to active points in the configuration C_1, the value of Δ_{ji} is determined by the relative ordering from C_1 ($\delta(c)$ is obtained in a similar way).

We retain the mutual absolute ordering of variables t^j and basic constants the same as in C_1 and move z at t^i. The location of t^j and c with respect to new active points is determined by $\delta(t^j)$ ($\delta(c)$ respectively) (if $\delta(t^j)=a$, then t^j is to be located between $z+a$ and $z+a+1$ in C_2, besides, if $t^j_\Delta=t^i_\Delta$ in C_1, then t^j must be placed just at $z+a$). If $t^i \le z+c_{max}$ in the configuration C_1, then we also insert the old value $z+c_{max}$ (we denote it further by z^0) in the configuration under construction (its location with respect to new active points is determined by $\delta(z^0) = c_{max} - [t^i-z] + \Delta_{0i}$, where $\Delta_{0i} = 0$, if $t^i_\Delta = 0$, and $\Delta_{0i} = -1$, if $t^i_\Delta > 0$).

Step 2 (deterministic).

While constructing the new relative configuration, at first we represent relative offsets only for those variables and constants which were within $[t^i, z+c_{max}]$ in the configuration C_1, as well as z^0_Δ (the relative offset of value z^0). We reorganize the relative ordering from C_1 the following way:

$$0 = t^i_\Delta < t^u_\Delta < \ldots t^r_\Delta \; < \; t^s_\Delta = z^0_\Delta \; < \; t^m_\Delta < \ldots < t^k_\Delta$$

those t^j_Δ which were greater than t^i_Δ in C_1 are represented here in ascending order;	if there was $t^s_\Delta = 0$ in C_1;	those t^j_Δ which were less than t^i_Δ in C_1 are represented here in ascending order.

Step 3 (nondeterministic).

Now we determine the place in C_2 for the variables located to the right of $z+c_{max}$ in C_1. Let us denote them in the ascending order by $t^{j_1}, t^{j_2}, t^{j_3}, \ldots, t^{j_s}$ (if two or more variables are equal, denote them by the same symbol t^{j_r}).

We allocate these variables in the same order, every one in an arbitrary admissible place in the configuration built so far. The variable $t^{j_{r+1}}$ must be allocated to the right of t^{j_r} allocated in the previous step (t^{j_1} is to be allocated to the right of z^0 (to the right of t^i if there was $t^i > z+c_{max}$ in the configuration C_1)).

For every variable t^{j_r} to be allocated we determine both its relations to the new active points (they are defined within Step 1 of the algorithm) and the place of its relative offset $t^{j_r}_\Delta$ in the relative configuration built so far (cf. the case of command $(X \to t^i)$ with exit "+" for the description of admissible locations of variable offsets).

When all variables $t^{j_1}, \ldots, t^{j_s}$ are allocated between active points or some t^{j_r} is located on the right of $z+c_{max}$ (the last active point in the configuration under construction) we delete z^0 and z^0_Δ from the obtained absolute and relative orderings, respectively, and stop the construction, retaining the mutual ordering of the variables on the right of $z+c_{max}$ the same as it was in C_1.

Examples. Let $C_1 = (0 < 2 < z < z+1 < t^1 < z+2 < t^2 < z+3 < t^3, 0 < t^2_\Delta < t^1_\Delta)$;

1) if $k = (2 \to t^1)$, then $C_1 \text{-}(k,\varepsilon)\!\to C_2$ iff

$C_2 = (0 < t^1 = 2 < z < z+1 < z+2 < t^2 < z+3 < t^3, 0 < t^2_\Delta)$;

2) if $k = (t^1 < t^2)$ and ε="+", then C_1 -(k,ε)→ C_2 iff $C_2 = C_1$, if ε="-", then the relation holds for no C_2;

3) if $k = (t^1 \dot{\to} z)$ and ε="+", then C_1 -(k,ε)→ C_2 holds iff C_2 is one of the following:

$0 < 2 < z = t^1 < t^2 < z+1 < t^3 < z+2 < z+3 , 0 = t^1_\Delta < t^3_\Delta < t^2_\Delta$;

$0 < 2 < z = t^1 < t^2 < z+1 < t^3 < z+2 < z+3 , 0 = t^1_\Delta < t^2_\Delta = t^3_\Delta$;

$0 < 2 < z = t^1 < t^2 < z+1 < t^3 < z+2 < z+3 , 0 = t^1_\Delta < t^2_\Delta < t^3_\Delta$;

$0 < 2 < z = t^1 < t^2 < z+1 < t^3 = z+2 < z+3 , 0 = t^1_\Delta = t^3_\Delta < t^2_\Delta$;

$0 < 2 < z = t^1 < t^2 < z+1 < z+2 < t^3 < z+3$, all possible t_Δ orderings for which there hold $t^1_\Delta = 0, t^2_\Delta > 0, t^3_\Delta > 0$;

$0 < 2 < z = t^1 < t^2 < z+1 < z+2 < t^3 = z+3 , 0 = t^1_\Delta = t^3_\Delta < t^2_\Delta$;

$0 < 2 < z = t^1 < t^2 < z+1 < z+2 < z+3 < t^3 , 0 = t^1_\Delta < t^2_\Delta$.

In this case the transformation algorithm after Step 2 yields the configuration
$0 < 2 < z = t^1 < t^2 < z+1 < z^0 < z+2 < z+3 , 0 = t^1_\Delta < z^0_\Delta < t^2_\Delta$.

Inserting t^3 in it in arbitrary place to the right of z^0, we obtain the variety of the abovementioned configurations C_2.

From the definition of relation -(k,ε)→ there follows

LEMMA 1. *For every statement k and its exit ε the relation C_1-(k,ε)→C_2 holds iff there are rationals $z,t^1,...,t^s$ satisfying the configuration C_1, such that the statement k can be executed at these values of variables with exit ε in a way that the tuple of variable values obtained in the result would satisfy the configuration C_2.*

Let α be an initial path in the program P. The *variable state* Q(α) corresponding to path α is defined by induction:

(1) to empty path there corresponds one element set containing the configuration with all internal variables equal to 0;

(2) if state Q(α) corresponds to path α, then for path α+(k,ε) we define Q(α+(k,ε)) $= \{ C' \in \mathcal{C}_P^{Var} \mid \exists C \in Q(\alpha): C$ -(k,ε)→ C' }.

The *state of input tapes* D(α) corresponding to path α is defined as a tuple $(d_1,d_2,...,d_m)$ (m is the number of input tapes), where d_i=0, if input statement from the i-th tape with exit "-" has occurred in the path, d_i=1 otherwise. The *(complete) state* S(α) corresponding to path α is defined as the pair (Q(α),D(α)).

Let us note the following. If we define for an arbitrary state $\sigma = (\sigma^Q,\sigma^D)$ ($\sigma^Q \subseteq \mathcal{C}_P^{Var}$, $\sigma^D \in \{0,1\}^m$) and path α (not necessarily initial) the conditional state S(σ,α) in a way similar to S(α) (only induction basis has to be changed), we see that an analogue of Lemma 4 from Theorem 1 holds.

The complete state S(α) defined for every path α allows us to construct (in a similar way as in the proof of Theorem 1) the reachability graph for the given program P.

The vertices of the graph are pairs (n,S) where n is a statement label and S=(Q,D) is a state of the program P ($Q \subseteq \mathcal{C}_P^{Var}$, $D \in \{0,1\}^m$). For every path α+(k,ε) in the program P we draw in the graph an edge labeled by ε from (k,S(α)) to (k',S(α+(k,ε)) if there is an edge labeled by ε from k to k' in the program P. It is easy to see that the

reachability graph is finite for every program P (since the set C_P^{Var} is finite).

For every initial path ν=($(n_0,S_0)\varepsilon_0$, $(n_1,S_1)\varepsilon_1$, ... ,$(n_r,S_r)\varepsilon_r$) in the reachability graph the corresponding path α=($n_0\varepsilon_0$, $n_1\varepsilon_1$, ... ,$n_r\varepsilon_r$) in the program P is called the *projection* of the path ν.

LEMMA 2. *An initial path α in program P is feasible iff there is an initial path ν in the reachability graph of P whose projection is α (cf. Lemma 6 in Theorem 1).*

If path α=($n_0\varepsilon_0$, $n_1\varepsilon_1$, ... ,$n_r\varepsilon_r$) is feasible, let us consider a fixed test T which forces the execution of this path. For every j=0,1,...,r+1 we consider the configuration of program variables C_j and state of tapes D_j after the path α_j=($n_0\varepsilon_0$, $n_1\varepsilon_1$, ... , $n_{j-1}\varepsilon_{j-1}$) while the program is executed on T. It follows from Lemma 1 and properties of D_j that there is a path ν=($(n_0,S_0)\varepsilon_0$, $(n_1,S_1)\varepsilon_1$, ... ,$(n_r,S_r)\varepsilon_r$) with the edge ε_r leading from (n_r,S_r) to (n_{r+1},S_{r+1}) in the reachability graph, such that $S_j = (Q_j,D_j)$ and $C_j \in Q_j$ for j=0,..,r+1.

Now let us prove that for every initial path in the reachability graph the corresponding projection is feasible in the program P.

Let us choose an initial path ν=($(n_0,S_0)\varepsilon_0$, $(n_1,S_1)\varepsilon_1$, ... ,$(n_{r-1},S_{r-1})\varepsilon_{r-1}$) in the reachability graph and consider the sequence of configurations $C_0,C_1,...,C_{r-1}$ such that

(1) $C_j \in Q_j$ for every j=0,..r-1;

(2) C_j -(k,ε)$\rightarrow$ C_{j+1} for every j=0,..r-2.

Let an edge labeled by ε_{r-1} leads from (n_{r-1},S_{r-1}) to (n_r,S_r) in the reachability graph, let $C_r \in Q_r$ and C_{r-1} -(k,ε)$\rightarrow$ C_r.

For every $j \le r$ we define ν_j to be the initial fragment ($(n_0,S_0)\varepsilon_0$,..., $(n_{j-1},S_{j-1})\varepsilon_{j-1}$) of ν. Let α_j be the projection of ν_j (α_j=($n_0\varepsilon_0$,$n_1\varepsilon_1$,...,$n_{j-1}\varepsilon_{j-1}$))

Let us prove by induction on j that for every ν_j there is a test T_j forcing the execution of P along α_j in a way that for every $s \le j$ the program variables after the path α_s satisfy the configuration C_s.

The existence of the test T_0 for empty path is obvious (test with empty input tapes A,B,...,C suffices).

The transition j $\rightarrow$ j+1 is different depending on n_j and ε_j:

1) If $n_j \in$ { c$\rightarrow t^i$, $t^u \rightarrow t^i$, z+c$\rightarrow t^i$}, or n_j= (X$\rightarrow t^i$) and ε_j="-", we can take $T_{j+1}=T_j$.

2) If n_j = ($t^i < t^u$), or n_j = ($t^i \overset{+}{\rightarrow}$z) and ε_j="-", then it follows from the definition of -(k,ε)$\rightarrow$ that every tuple of variables satisfying C_j forces the execution along the branch needed, once again we can take $T_{j+1}=T_j$.

3) If n_j= (X$\rightarrow t^i$) and ε_j="+", then by the definition of reachability graph no input statement from tape X with exit "-" has occurred in path α_j. The test T_{j+1} is obtained from T_j concatenating to tape X a cell with a value that locates t^i in the configuration C_{j+1} in the appropriate place with respect to other variables and constants of the program.

4) If n_j= ($t^i \overset{+}{\rightarrow}$z) and ε_j="+", we describe a method for obtaining T_{j+1} from T_j . Let us fix the values of program variables in the moment when P has executed the path α_j on the test T_j. Consider variables with the values fixed greater than z+c_{max} (the list of them can also be obtained from the configuration C_j). Each of these variables has received its current value in path α_j only by means of input statement (because they are greater

than $z+c_{max}$) and has used this value in various comparison statements. So each of these values has an input tape cell corresponding to it in the test T_j. Let us update the values of cells of input tapes corresponding to the considered variables which are located below $z+c_{max}$ (or are equal to it) in the configuration C_{j+1}, in order to ensure that these variables after the execution of n_j have correct places with respect to active points and other variables' relative offsets in C_{j+1}.

It follows from the definition of relation $-(k,\varepsilon)\rightarrow$ in the case of positive assignment statement, namely, Step 3 of the algorithm, that we shall be able to update the values of tape cells preserving for the considered variables t^u the ordering and locations with respect to the old $z+c_{max}$. It means that on the new test T_{j+1} the program traverses the path $\alpha_{j+1}=\alpha_j+(n_j,\varepsilon_j)$, besides, for $s\leq j+1$, the internal variables satisfy the configuration C_s after the path α_s.

This proves the lemma.

We note that the proof of the lemma yields a constructive method how to find for a feasible path a test executing it. So, just as in the case of Theorem 1, we can build a finite CTS from the reachability graph of the program.

This completes the proof of the theorem.

Theorem 11 yields a principal possibility to build CTS for every program in $\mathbf{L_7}$. However, if a program has a large number of variables, the algorithm given in the proof can be infeasible in practice due to enormous number of variable configurations. Let us describe a principally more efficient algorithm of CTS construction for programs in $\mathbf{L_7}$ based on the usage of inequality systems.

We begin with the association (just as for programs in $\mathbf{L_0}$) of inequality system $N(\alpha)$ to every initial path α, such that

1) path α is feasible iff $N(\alpha)$ has a rational solution,

2) every solution of $N(\alpha)$ with respect to cells of input tapes yields a test on which the program executes the path α.

<u>Example.</u> Let us consider the path

$(A \rightarrow t^1 +, t^1 \stackrel{+}{\rightarrow} z +, z+5 \rightarrow t^2, A \rightarrow t^1 +, t^1 < t^2 +, t^1 \stackrel{+}{\rightarrow} z +)$.

Its inequality system is

$$\begin{cases} t_0{}^1 = t_0 = z_0 = 0 \\ t_1{}^1 = A_1 \\ t_1{}^1 > z_0 \,,\, z_1 = t_1{}^1 \\ t_1{}^2 = z_1 + 5 \\ t_2{}^1 = A_2 \\ t_2{}^1 < t_1{}^2 \\ t_2{}^1 > z_1 \,,\, z_2 = t_2{}^1 \end{cases}$$

It is easy to see that with respect to A_1 and A_2 it is equivalent to the system $0 < A_1 < A_2 < A_1 + 5$, the consistency of which assures the feasibility of the path.

Path inequality systems are used for efficient obtaining for a given path in the reachability graph a test forcing the execution of the corresponding path in the program.

The construction of the reachability graph is based on relating a set $\mathcal{U}(\alpha)=\{U_1(\alpha),...,U_r(\alpha)\}$ of certain type inequality systems $U_s(\alpha)$ to an arbitrary path α in

the program. So the vertices of reachability graph are pairs $(n_i,\mathcal{U})$, where n_i is a statement label and $\mathcal{U}=\mathcal{U}(\alpha)$ for some path α. The idea of inequality systems $U_s(\alpha)$ is that the set $Q(\alpha)$ of configurations, corresponding to α, will be distributed during the graph construction in several subsets $Q_1(\alpha),Q_2(\alpha),\dots,Q_r(\alpha)$, such that their union coincides with $Q(\alpha)$ and each $Q_i(\alpha)$ is coded by the inequality system $U_i(\alpha)$ (actually, the state of input tapes $D(\alpha)$ is represented in every $U_s(\alpha)$ as well).

When constructing the reachability graph we use the systems consisting of inequalities of the form

(1) $a\lambda(b+c_1)$, where $\lambda\in\{>,<,\geq,\leq\}$, a and b are program internal variables or basic constants; $0\leq c_1\leq c^1_{max}$,

(2) $A < 0$, $A > 0$, where A is a program input tape (external variable).

Let us call the systems of the described form *state inequality systems.*

Let us say that a state inequality system U *codes* a pair (Q,D), where $Q\subseteq \mathcal{C}_P^{Var}$, $D\in\{0,1\}^m$ (m is the number of program input tapes) if

(1) every solution of U with respect to z, t^1 ,..., t^k satisfies at least one configuration $C\in Q$;

(2) for every configuration $C\in Q$ there are values of variables z, t^1 ,..., t^k satisfying both C and U;

(3) for every tape A, U contains inequality $A<0$ iff $d_A=0$ in the state D; there is no inequality of the form $A>0$ in U.

Let us represent the state inequality systems as graphs (just as it was done in the case of the base language $\mathbf{L_0}$). Let all program internal variables, basic constants and input tape names serve as vertices in them. We label the edges of the graphs by weights of the form (x,ξ), where $x\in \mathbf{Z}$ and $\xi\in\{0,"+"\}$ (the comparison and the addition can be defined in the set of weights quite naturally: $(x_1,\xi_1)>(x_2,\xi_2)$ if $x_1>x_2$ or $x_1=x_2$ and $\xi_1="+"$, $\xi_2=0$; $(x_1,\xi_1)+(x_2,\xi_2)=(x_1+x_2,\xi_1+\xi_2)$, where $\xi_1+\xi_2=$ "+" if either ξ_1 or ξ_2 is "+", $0+0=0$).

Let us represent the inequalities of the system in the graph the following way:

$a\geq b+c_1$ - an edge, weighted by $(c_1,0)$ from a to b,

$a>b+c_1$ - an edge, weighted by $(c_1,"+")$ from a to b,

$a\leq b+c_1$ - an edge, weighted by $(-c_1,0)$ from b to a,

$a<b+c_1$ - an edge, weighted by $(-c_1,"+")$ from b to a.

Not difficult to prove that the state inequality system has a solution iff there is no cyclic path with positive sum of weights and no path from constant vertex c_0^1 to c_0^2 with the weight greater than $(c_0^1-c_0^2,0)$ in its graph.

We define the exclusion of a graph vertex a as replacement of all the edges leading to and from a by edges bd between other vertices b and d, such that before exclusion the graph contained both the edges ba and ad. The weight w(bd) is defined as the sum of weights w(ba)+w(ad). For every two vertices b and d we retain only that edge from b to d which has the maximum weight.

We define an edge a_1a_n with weight $w(a_1a_n)$ in the state inequality systems graph (further - state graph) *reducible* if there are vertices $a_2,\dots,a_{n-1}$ ($n\geq3$), such that the graph contains edges a_ia_{i+1}, $i=1,\dots,n-1$ and the sum of weights $w(a_1a_2)+ \dots + w(a_{n-1}a_n) \geq \geq w(a_1a_n)$.

We introduce a *reduced form* for each state graph, which can be obtained from the graph the following way:

(1) if the state graph is contradictory (i.e., it is a graph of a contradictory inequality system), replace it by the inequality systems "0>0" graph, otherwise execute (2), (3) and (4);

(2) for every cyclic path $a_1a_2,...,a_na_1$ with (0,0) sum of weights, find out the ordering $a_{j^1}\lambda_1 a_{j^2}\lambda_2 ... \lambda_{n-1} a_{j^n}$, ($\lambda_j \in \{<, \leq\}$) of vertices $a_1,a_2,...,a_n$. Then replace all edges between these vertices by the edges from a_{j^s} to $a_{j^{s+1}}$ and from $a_{j^{s+1}}$ to a_{j^s} (s=1,..,n-1), weighted by $(a_{j^s}-a_{j^{s+1}},0)$ and $(a_{j^{s+1}}-a_{j^s},0)$ respectively;

(3) for every cyclic path with (0,0) weights on all its edges, gather all its vertices into one vertex;

(4) delete the reducible edges from the graph until it contains no ones.

Let us call a variable or basic constant a in a state inequality system U *very essential* if there exists an edge leading to or from a, labeled by the weight different both from (0,0) and (0,"+") in the reduced form of U graph. We denote the set of very essential variables and basic constants in U by VE(U).

Let us consider an arbitrary set V of program P variables and basic constants. We say that a state inequality system U is *complete* with respect to V if for every $a,b \in V$, such that $a \leq b < z$ holds in one of U solutions (z is the value of program real time counter), the inequalities $a \leq b$ and $b < z$ hold in every solution of U.

In other words, the complete with respect to V state inequality system must unequivocally determine the ordering of set V variables and constants which are below than z. We define U-bottom(V)=$\{a \in V \mid a<z$ in some solution of U$\}$.

Now we are able to describe the reachability graph construction algorithm.

We relate the state inequality systems' set with just one element, namely, the inequality system $0 = z = t^1 = ... = t^k$ to the empty path.

Assume that we have constructed the state inequality system set $\mathcal{U}(\alpha)=$ $=\{U_1(\alpha),U_2(\alpha),...,U_n(\alpha)\}$ corresponding to a path α. Let us describe how to construct the set $\mathcal{U}(\alpha+(k,\varepsilon))$ corresponding to the path $\alpha+(k,\varepsilon)$.

The following cases are possible:

1) $k=(t^i \dot{\rightarrow} z)$ and ε="-", or $k=(t^i<t^j)$.

Add the corresponding inequality to each of $U_s(\alpha)$.

2) $k=(c \rightarrow t^i)$ or $k=(t^j \rightarrow t^i)$, or $k=(z+c \rightarrow t^i)$.

Exclude t^i from each $U_s(\alpha)$. Add the equality $c=t^i$ ($t^j=t^i$ or $z+c=t^i$ respectively) to the obtained systems.

3) $k=(X \rightarrow t^i)$ and ε="+".

Exclude t^i from each $U_s(\alpha)$. If any of $U_s(\alpha)$ contains the inequality $(X<0)$, add the inequality $(X>0)$ to it.

4) $k=(X \rightarrow t^i)$ and ε="-".

If $U_s(\alpha)$ does not contain the inequality $(X<0)$, add it to the system (s=1,..,r).

5) $k=(t^i \dot{\rightarrow} z)$ and ε="+". Process every $U_j(\alpha)$ the following way:

Add the inequality $(z<t^i)$ to $U_j(\alpha)$, exclude z from the obtained system. Locate z at t^i (add the equality $z=t^i$ to the system). If the obtained system $U'_j(\alpha)$ is not complete with respect to the variables' and constants' set $VE(U'_j(\alpha))$, supplement it (by determining the ordering of variables and constants in U-bottom($VE(U_j(\alpha))$)) in all possible ways to the complete ones, so obtaining the list $U'_{j,1}(\alpha),U'_{j,2}(\alpha),...,U'_{j,p}(\alpha)$ of the complete systems

(each $U'_{j,s}(\alpha)$ is a complete system due to $VE(U'_j(\alpha))=VE(U'_{j,s}(\alpha))$.

Further process every $U'_{j,s}(\alpha)$ the following way:

Exclude from it all variables and constants being in U-bottom($VE(U_{j,s}(\alpha))$) (we have defined the exclusion of the graph vertex, it can be applied also to the constant one). Afterwards add the inequalities, determining the ordering of the set U-bottom($VE(U_{j,s}(\alpha))$) variables and constants and their relations with z, to the obtained system, so we have obtained an element of the set $U(\alpha+(k,\varepsilon))$.

The transformation of the state inequality systems in the case of the positive assignment statement with the exit "+" may be roughly considered as erasing weights on edges connecting the variables and constants which are below z (i.e., represent the past time moments) one with another and to other program internal variables and constants. However, in the general case a more sophisticated approach (like the performed one) is necessary. We note also that in the most typical cases the number of systems in $\mathcal{U}(\alpha)$ is quite small; for the most of real time systems for every α the set $\mathcal{U}(\alpha)$ consists of just one system $U(\alpha)$ which codes the set of configurations $Q(\alpha)$.

Example. Let us consider the path

$(A \to t^1 +, t^1 \dot{\to} z +, z+5 \to t^2 ,A \to t^1 +, t^1 < t^2 +, t^1 \dot{\to} z +)$.

For it $\mathcal{U}(\alpha_0) = \{U(\alpha_0)\} = (0 = z = t^1 = t^2)$,

$\mathcal{U}(\alpha_1) = \{U(\alpha_1)\} = (0 = z = t^2)$,

$\mathcal{U}(\alpha_2) = \{U(\alpha_2)\} = (0 = t^2 < z = t^1)$,

$\mathcal{U}(\alpha_3) = \{U(\alpha_3)\} = (0 < z = t^1 < z+5 = t^2)$,

$\mathcal{U}(\alpha_4) = \{U(\alpha_4)\} = (0 < z < z+5 = t^2)$,

$\mathcal{U}(\alpha_5) = \{U(\alpha_5)\} = (0 < z < z+5 = t^2 , t^1 < t^2)$,

$\mathcal{U}(\alpha) = \mathcal{U}(\alpha_6) = \{U(\alpha_6)\} = (0 < z = t^1 < t^2 < z+5)$.

Using the state inequality systems sets $\mathcal{U}(\alpha)$ as states corresponding to program paths we build a reachability graph for the program likewise in the proof of Theorem 11.

LEMMA 3. *An initial path α in the given program P is feasible iff there is an initial path v in the constructed reachability graph whose projection is α.*

We note that for every path α all weights on edges of state graphs for inequality systems $U_1(\alpha),\ldots,U_r(\alpha)$ are bounded from $(-c_{max},0)$ to $(c_{max},"+")$, this ensures us about the finiteness of the constructed graph.

The proof of the lemma is based on inductive demonstration that for every path α the inequality systems $U_1(\alpha),\ldots,U_r(\alpha)$ code the pairs $(Q_1(\alpha),D(\alpha)),\ldots,(Q_r(\alpha),D(\alpha))$, respectively, where the union of configuration sets $Q_s(\alpha)$ coincides with $Q(\alpha)$.

The given construction of the reachability graph, together with the solvability of path inequality systems, form the base for the desired more efficient algorithm of CTS construction for the programs in $\mathbf{L}_7$.

At the end of the section we introduce a new programming language $\mathbf{L}_7^{\prime}$ with operations over both rational and integer data types. We define the language $\mathbf{L}_7^{\prime}$ as follows:

Every program in L_7' may have internal variables of two types - rationals (i.e., t^i and z) with operations of L_7 permitted and integers with permitted operations from language L_0. A type mismatch in the commands in L_7' is forbidden. Each external variable - input tape of the program has a definite type (integer or rational) as well. This determines in which system of commands the input from this tape can be used.

THEOREM 12. *There exists an algorithm constructing finite CTS for every program in* L_7'.

The proof of the theorem relies upon the construction of the reachability graph, where for every path α in the program corresponds the state $(S_0(\alpha), S_7(\alpha))$ with $S_0(\alpha)$ being the state in the sense of L_0, associated with α, but $S_7(\alpha)$ - the one in the sense of L_7.

The proof that an initial path in the program is feasible iff it is a projection of any path in the reachability graph follows from the analogous results for languages L_0 and L_7 (cf. Lemma 6 from Theorem 1, Lemma 2 from Theorem 11).

A more efficient generation of CTS for programs in L_7' can be performed by using the inequality systems, i.e., by ascribing the state $(S_0(\alpha), \mathfrak{U}(\alpha))$ to an arbitrary path α in the program.

Just as for programs in L_0 (cf. Section 4) the conditional programs can be introduced in L_7' as well (the conditions are allowed to stand for the variables of each data type separately). Likewise in Section 4 we can introduce the notions of the correct test and correct CTS for conditional programs in L_7'. Following the aforementioned ideas we can prove

THEOREM 13. *There exists an algorithm constructing finite correct CTS for every conditional program in* L_7'.

10 CTS Generation for Real Time Systems. An Example

In this section we consider a simple example how to apply the CTS generation means to real time programs.

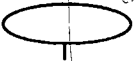
10.1 Example Specification Language

In the current section we use a subset of specification language SDL [16] to specify the example. Now let us describe the subset.

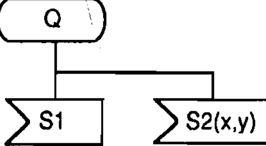
Only one SDL process is used to describe a real time system. SDL process is a program executing in real time and communicating with the environment by means of signals. SDL process has an input queue into which the environment at certain time moments puts input signals for the process (time moment can be an arbitrary rational, time counting begins at the process start). Process input signal can have a definite number of integer valued parameters, the signal is recorded in the queue together with its parameters. We can assume for sake of simplicity that no two signals are put into the queue simultaneously. The process also has output signals, these signals are sent to the environment at certain time moments according to the process program (process diagram), as a reaction to input signals.

SDL process is a finite state machine extended by variable notion and some special statements. To be more precise, we assume that SDL process can use a finite number of internal variables, the process diagram can contain the following statements.

1. START - the beginning of the process execution. We assume that all process internal variables are initialized to 0 at the execution of START. We depict the statement in the diagram the following way:



2. STATE / INPUT - the complex of statements for awaiting / reading of input signals, it has the following form in the process diagram:



Here Q is a state name, S1 and S2 are names of signals awaited in this state, x,y are process internal variables to which the values of parameters conveyed by signal S2 are assigned at consumption (reading) of S2.

If the process has reached the state Q during the execution, it is awaiting for the arrival of some signal in the input queue. At the moment when a signal arrives the signal is consumed (and the necessary assignments of parameter values to internal variables performed). Further control flow in the diagram depends on the name of incoming signal (for the sake of clarity we assume that reaction to every possible signal is specified in every state).

3. OUTPUT - signal sending statement. It has the form:

S(x)

and it denotes the sending of signal to the environment at the given moment of process execution. Here S is a signal name and x is an internal variable whose value is assigned to parameter of the signal.

4. SAVE - signal save statement, it is included in STATE / INPUT complex the following way:

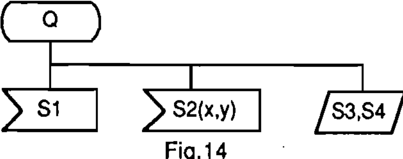


Fig.14

The location of signals S3 and S4 in SAVE statement at the state Q means that, if the process is in state Q, signals S3 and S4 are not consumed but retained in the input queue in the order of their arrival (i.e., the process waits for the arrival of some other signal, S1 or S2 in the case). For every state Q we assume that the name of every input signal is mentioned in just one INPUT or SAVE statement at this state.

If the process diagram contains a fragment (fig.15) and there the sequence S3(1), S4, S3(2), S1, S2(0,0) of signals arrive to the process queue, then these signals are consumed in the order S1, S3(1), S4, S3(2), S2(0,0) (we assume the process being in the state Q just before the arrival of signal S3(1)).

5. TASK - action statement representing assignment to internal variables of the process, e.g.,

x := y , x := 5

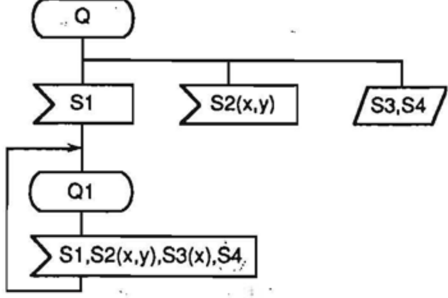
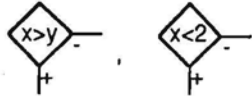


Fig.15

6. DECISION - representing variable comparison statement (in fact, the same comparison statement used in L_0):



7. SET, RESET statements and timer signals.

SDL process has a predefined function *now*, at every time moment returning the numeric time value of this moment (certain nonnegative rational). Process may have a finite number of timers (informally each timer is an "alarm-clock" which can be set to send a special signal after the expiring of a definite time interval).

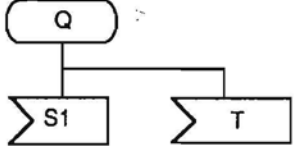
A timer in SDL process can be set by statement

set (now+c,T)

Here T is a timer name and c - an integer constant (a timer is said to be active after setting). The activity of the timer T, before it "rings", can be disrupted by the statement

reset (T)

When the interval of timer activity expires (i.e., c time units have passed) and it has not received the reset statement, a special signal is put into the process input queue, the signal name being the same as timer name. This signal can be consumed in a process state (a special input branch has to be added to the state):



If some active timer is set, an automatic reset is executed for the timer before the new setting. The statement "reset(T)" also erases all signals with the name T from the input queue (if there are such).

The execution of the process begins with START statement at a time moment *now* =0, further processing is performed in accordance with the process diagram. We assume that all internal actions of the process (assignment, comparison, signal sending/consuming, timer setting/resetting) are performed instantaneously, so the function *now* changes its value only when the process waits for signals (or timer) in some state.

10.2 Passenger Lift Specification

We describe a control program for some kind of passenger lift by means of SDL process. The environment for the process consists of lift users and lift hardware.

A lift user can press a call button in every floor thus sending the signal S with parameter x (the floor number) to the process. Besides that the user can press the button in the lift-cage to pass the request for the lift to go to some floor; so the signal R with one parameter - the destination floor number is sent to process. In some situations the user can also generate signals FU(FloorUp) and FD(FloorDown) by leaving the lift-cage and entering it respectively (i.e., changing the status of cage floor).

The lift hardware consists of lift driving motor controlled by signals M-Up,M-Down, M-Stop, lift door motor (controlled by signals MDoor1(open the door), MDoor2 (close the door) and MDoorStop) and some sensors informing the process about the physical state of the lift. The following signals from sensors to process are considered: Z(x) - floor number x is reached, DOp(Door is Open), DC(Door is Closed).

Behaviour of the lift can be characterized by the following:

1) the lift has no memory for user requests, signals S(x),R(x) are accepted for processing only after previous request has been executed,

2) if empty lift with open door stays in some floor for more than 20 seconds, the door is being closed,

3) if the status of cage floor is changed while the door is closing (i.e., somebody has entered or left the cage), the closing of the door is interrupted and the door opens. Besides, if the door was being closed to execute some request to go somewhere, the request is canceled.

Besides the control algorithm also a partial correctness check of incoming signals is included in the specification of lift process, it will enforce tests in the generated CTS to be actually possible sequences of lift input signals. To do this in the specification language some exits are allowed to be pending for branching statements (DECISION statements, STATE/INPUT complexes). This is done in a way similar to conditional programs in Section 4.

The specification of the lift process is presented in fig. 16.1 thru 16.3.

10.3 Simulation of SDL Process by Program in L'_7

By a test for an SDL process we understand a sequence of signals which are put by environment at certain time moments into the process input queue (every signal is considered together with its parameter values). We remind that simultaneous input signals are not allowed.

If signal S1 with parameters 7 and 12 is sent to the process at moment 3, signal S2 is sent at moment 3.7 and another signal S1 with parameters 0 and -5 at 7.22, then the sequence of signals is recorded as a test for SDL process the following way:

(S1(7,12) at 3), (S2 at 3.7), (S1(0,-5) at 7.22)

A test for SDL process is said to be correct if the process never reaches pending exit while executing on the test.

We don't consider direct construction of correct CTS for SDL processes. Instead we describe a method how to simulate specifications (programs) in the described subset of SDL by conditional programs in L'_7. We also demonstrate previously described algorithm for the construction of correct CTS on lift process example.

We say that a program P(R) in L'_7 simulates SDL process R if:

1) a one-to-one mapping between correct tests for process R and program P(R)

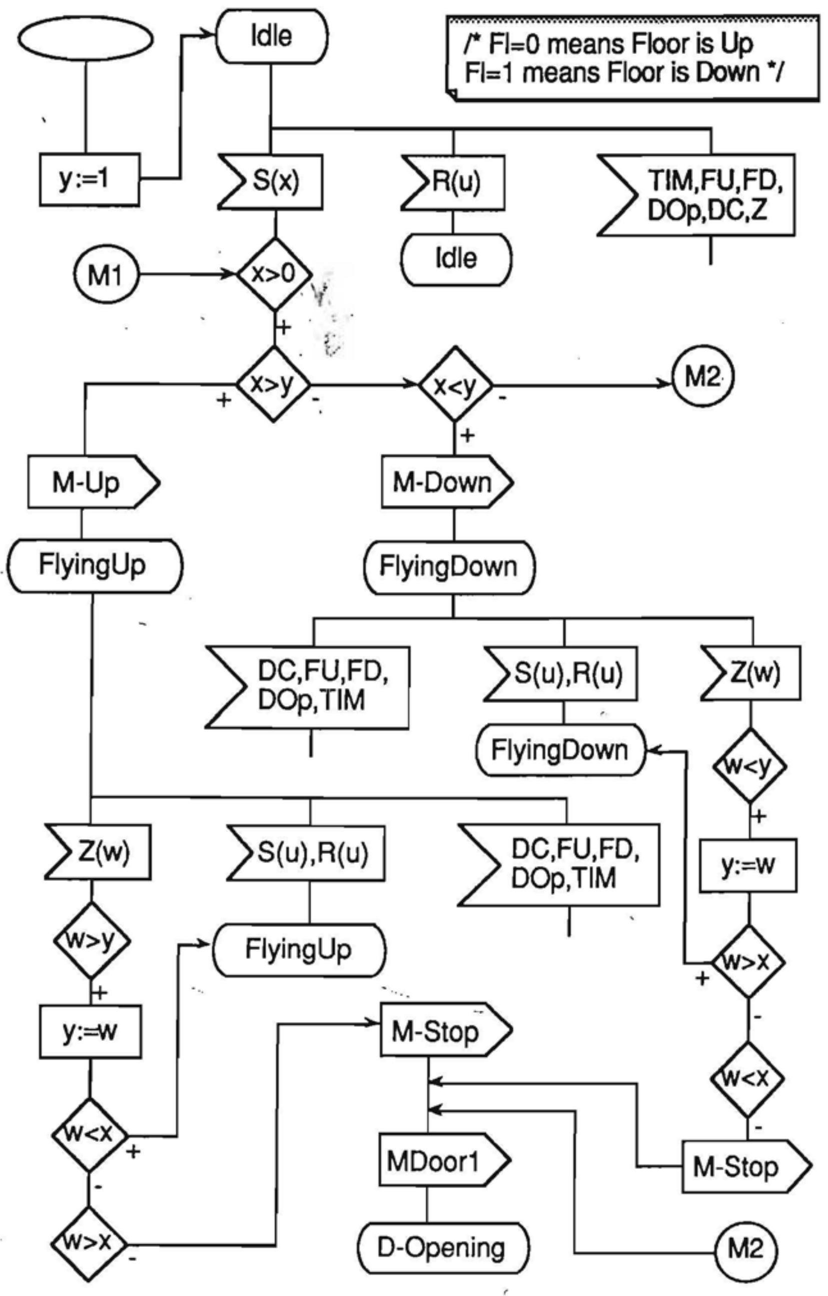


Fig.16.1

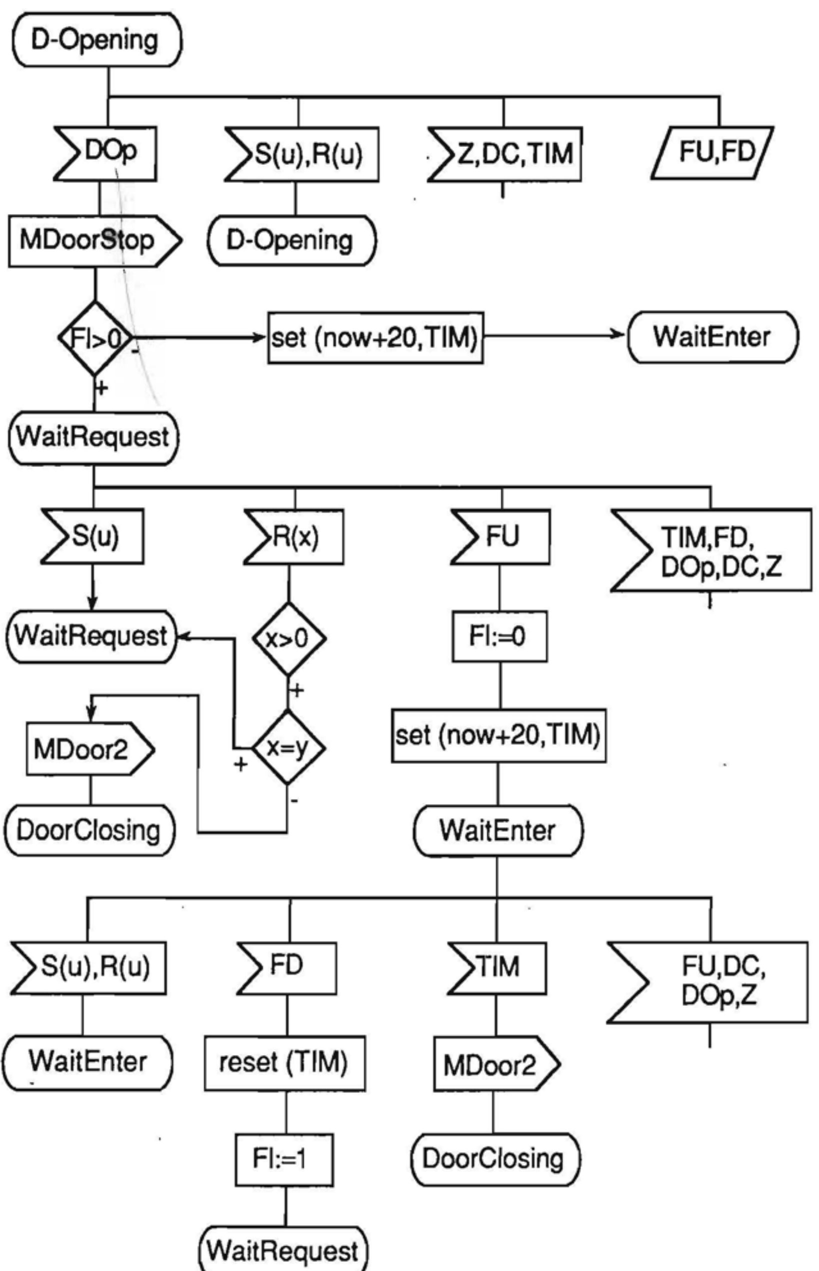


Fig.16.2

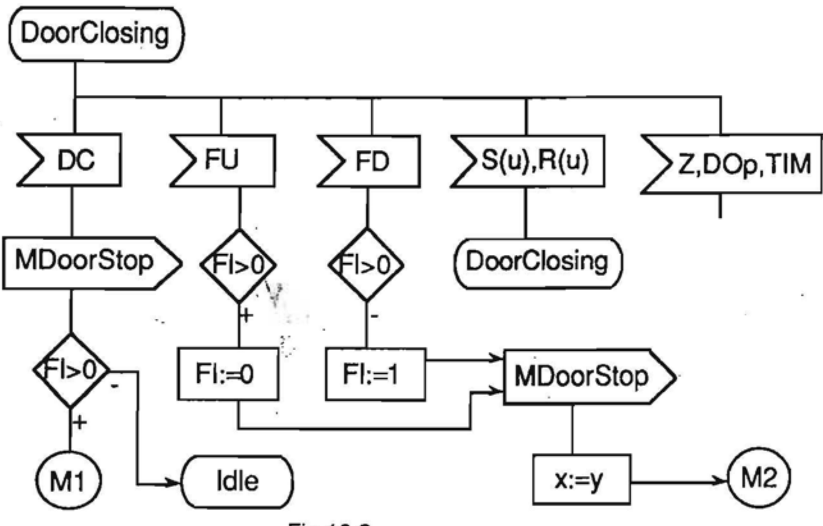


Fig.16.3

is defined together with algorithms yielding program test from the corresponding process test and vice versa,

2) for every $\mathcal{S}$=(V_1 ,V_2 ,...,V_a) being a correct CTS for program P(R), the set consisting of tests for process R corresponding to tests V_i is a correct CTS for process R according to some analogue of criterion C_1 .

Now let us describe a method how to transform a correct test V for SDL process R into a test for simulating program P(R).

The execution of process R on the test V means that at certain time moments the environment and process timers insert into the process input queue definite signals. Likewise, the test V determines the sequence in which the process R reads (consumes) the signals from the queue (this sequence can differ from insertion sequence due to SAVE statements). Relying on this we write the test for P(R) corresponding to the test V on three tapes T,S, and P the following way:

on the tape T we write the arrival time for every signal read by process R;

on the tape S we write the signal name coded by natural;

on the tape P we write the signal parameter values (if the signal has parameters).

Arrival times, signal names and parameter values are written on tapes in the signal reading sequence corresponding to the test V (hence there will be correct tests for P(R) with not increasing cell values on the tape T).

It is easy to see that a test for SDL process can be simply obtained back from the corresponding test for the simulating program. The fact that every correct test for the simulating program corresponds to a correct test for SDL process is guaranteed by the construction of simulating program described below.

The main idea of the simulation of SDL process performance on some correct test by program in $L_7^{\bullet}$ is to represent the current time (i.e., the value of the function *now*) by the real time counter z. Every time the process reads a new signal the simulating $L_7^{\bullet}$ program reads the arrival time of this signal from tape T into variable t and assigns it to the real time counter z by means of statement $t \xrightarrow{+} z$ ("-" exit from this statement will be processed depending on the situation, see below).

The simulating program in L_7' is obtained from SDL process diagram the following way:

1)START statement is transformed into the start label "—>" of the program;

2)STATE/INPUT statement complex (timer signals and SAVEs are considered later), fig.17,

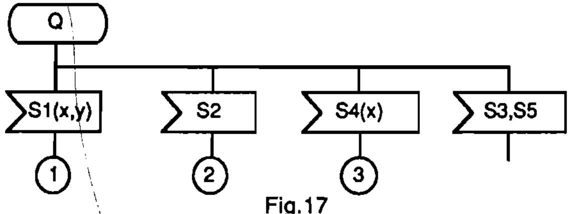


Fig.17

is transformed as shown in fig.18.

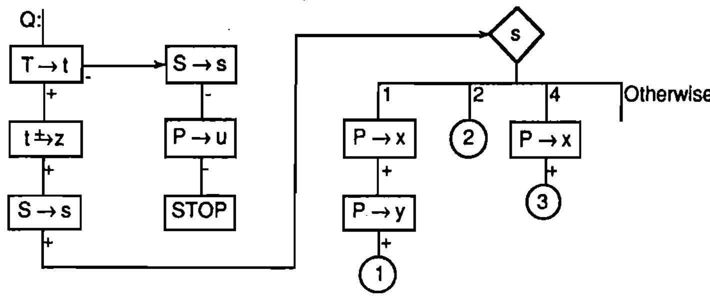


Fig.18

Here ◇s is a normal CASE statement (easily expressible in L_7'), further on we do not show the pending OTHERWISE branch, this branch sets correctness condition upon the code of signal name contained in the corresponding tape cell (s in the example cannot assume the value either 3 or 5, or some other value different from 1,2 and 4);

3) output signals are not represented in L_7' -program (they are inessential from CTS viewpoint);

4) internal variables are transferred to L_7' -program without changes, only the syntactic form of variable operations is changed (see the example below);

5) if SDL process has at least one SAVE statement, then in the simulating L_7' -program:

(i)for every signal used in at least one SAVE statement the variable t^s (s being the code of signal name) is introduced;

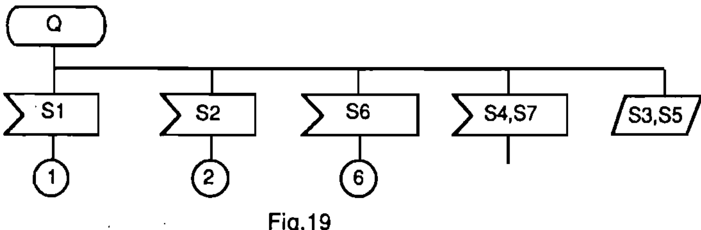


Fig.19

(ii)for every state Q in SDL process, e.g., the state shown in fig.19, (including

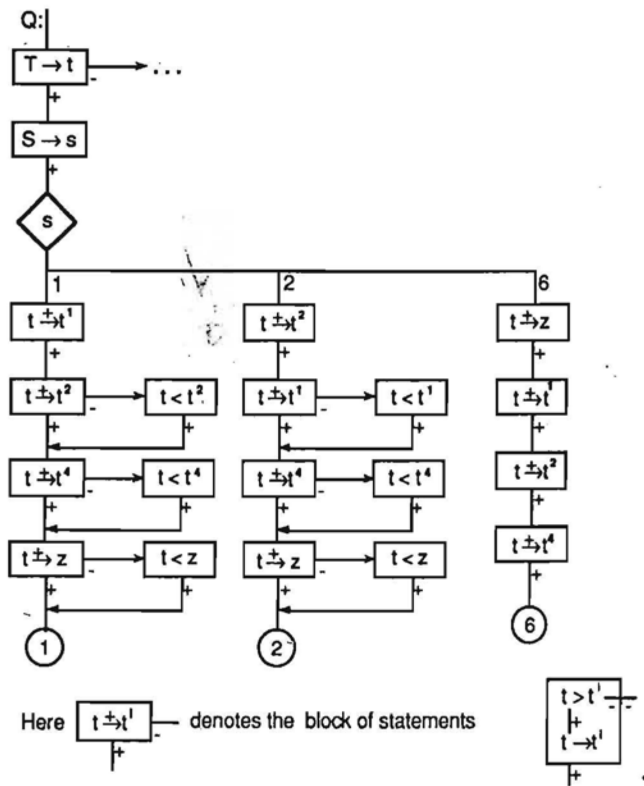


Fig.20

states without SAVE statements), if SAVE statements in the process contain, let us say, signals S1,S2,S3,S4,S5 and do not contain signals S6,S7, then the corresponding L_7' -fragment is transformed as shown in fig. 20.

If some signals Si,Sj,Sk appear in SAVE statements of the process always together, sole variable t^i can be defined for all of them.

If a signal S in the simulated SDL-process appears in some SAVE statement, then the arrival time of the signal, read from the input tape T, may happen to be less than the current value of z (i.e., less than the corresponding value of the function *now* in the process) because the signal could have been retained in the input queue for some time. In the given moment of execution of L_7' -program the value of variable t^s, corresponding to S indicates the lower bound for the arrival time of S (t^s is the largest arrival time for the signals read so far in the states which don't contain S in their SAVE statements).

The reading of signal S with the arrival time less than t^s would violate the FIFO discipline of the input queue (taking into account the corrections made by SAVE's).

6) for every timer Tn in the process we define a corresponding variable t^n in the simulating program. In the situation of timer Tn being active the variable t^n will hold the value of the expected moment of signal appearance from the timer; let t^n=-1, if Tn is inactive (for the sake of simplicity we don't consider the case when timer signals are retained by SAVE statements in SDL process, principal complications do not appear in

this case, too).

If SDL process has timers, e.g., T1 and T2, then every "-" exit from statement reading the tape T is augmented by condition expressing the inactivity of the timers:

$T \rightarrow t$ — $t^1 = -1$ + $t^2 = -1$ + ...

Fig.21

If input of timer signals, e.g., T3 and T4 is admissible in the state Q of the process

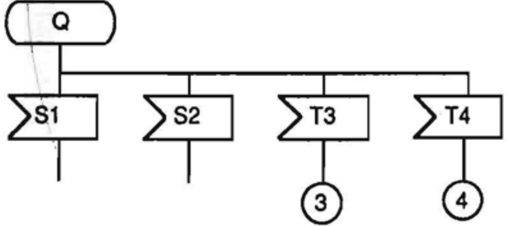


Fig.22

(we define that input of timer signal Tn is admissible in state Q if there is a path in the process diagram from START to Q such that the timer Tn remains active after the path), then the corresponding fragment (see fig. 22) is transformed in such way:

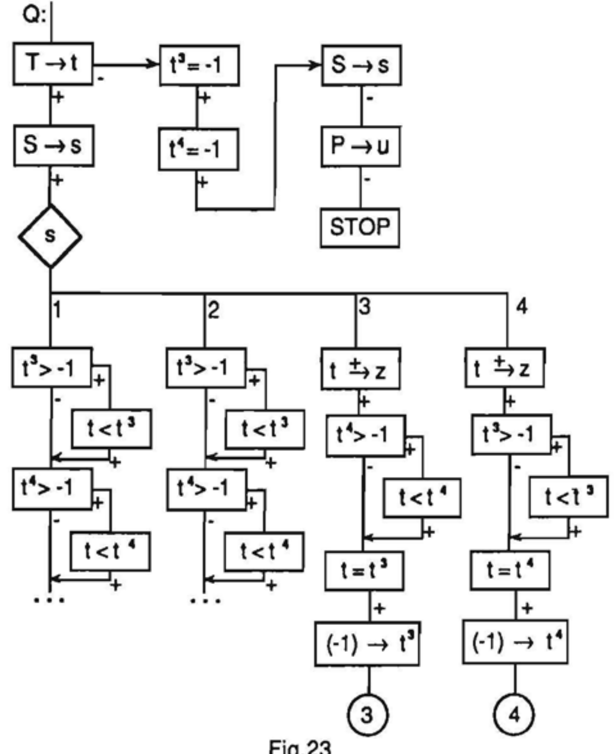


Fig.23

7) statement set(now+c,Tn) is transformed into $z+c \rightarrow t^n$, statement reset(Tn) into $(-1) \rightarrow t^n$.

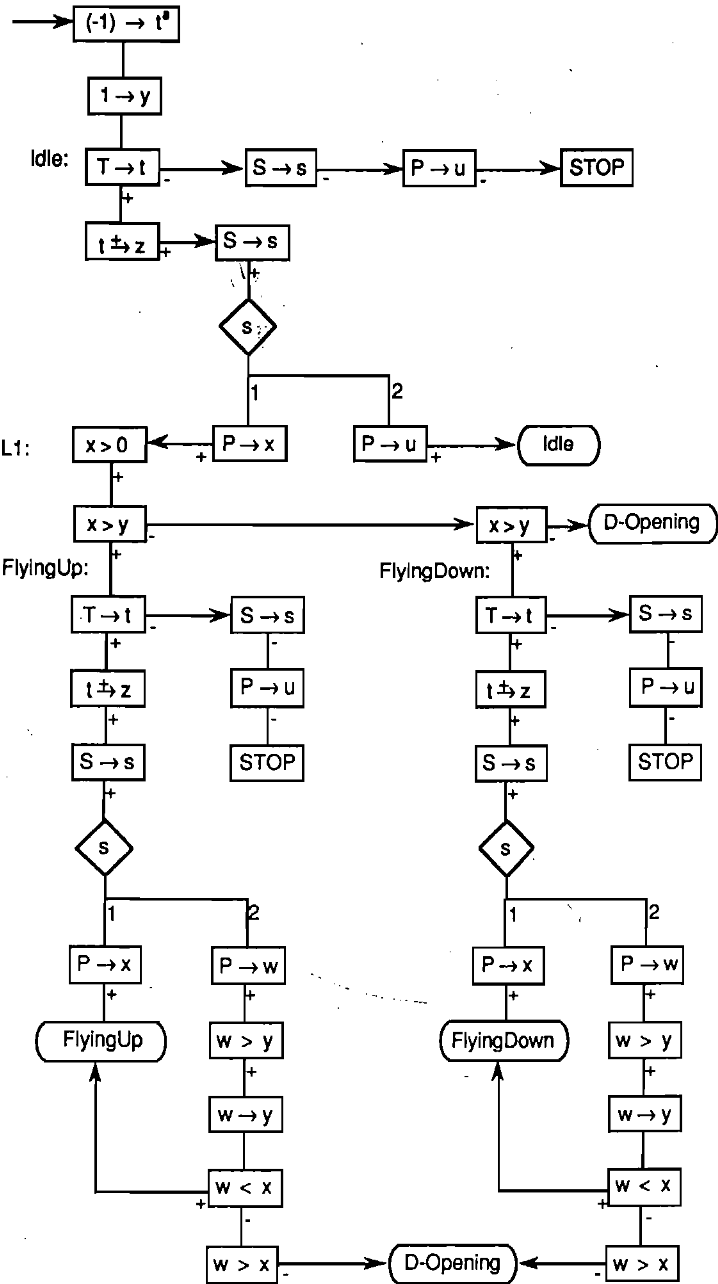


Fig.24.1

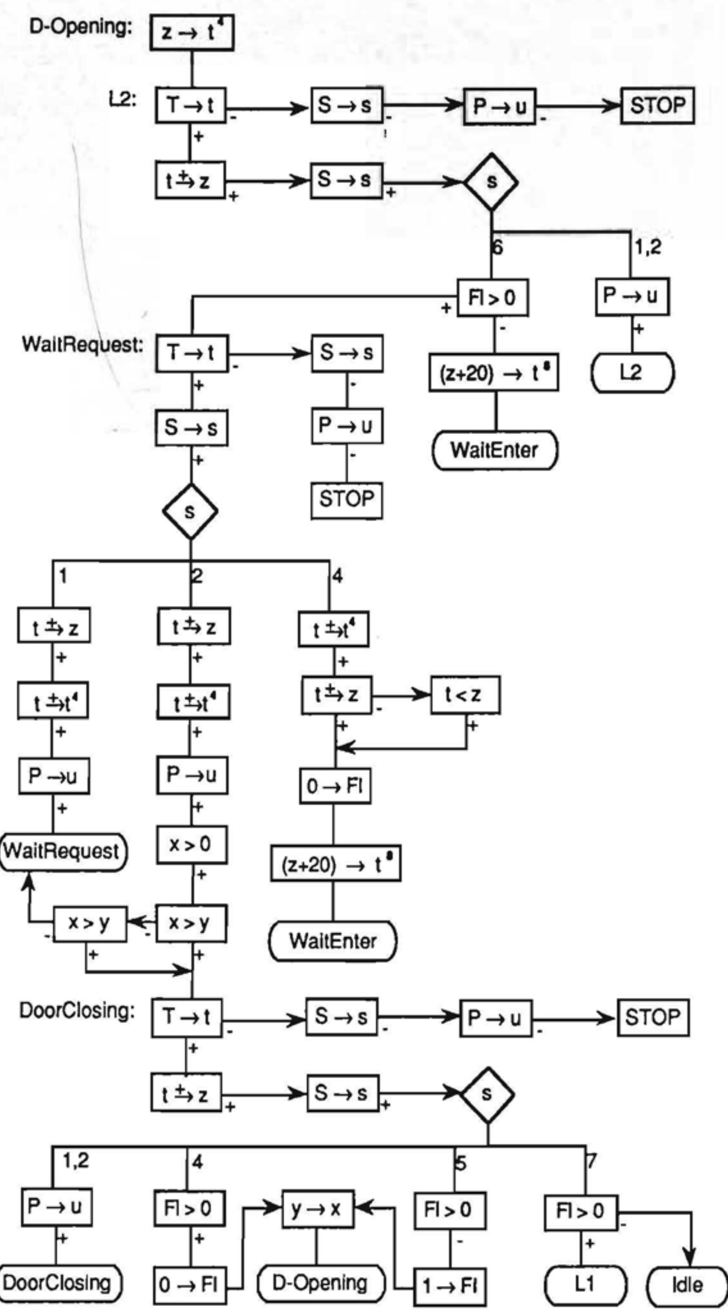


Fig.24.2

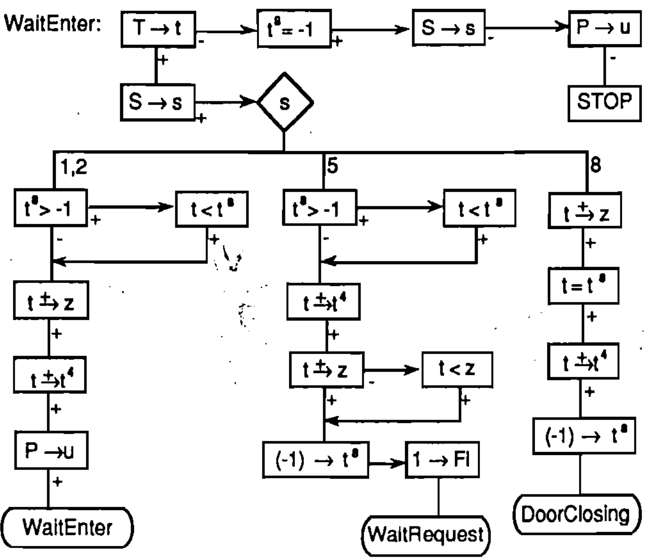


Fig.24.3

In order to reduce the size of the simulating L_7' -program obtained by the described algorithm we perform some simple optimizations with respect to rational internal variables (i.e., variables t^i and z) preserving the sequence of reads and the value of z at any read from the tape T on every correct test (see the example below).

10.4 Simulating Program for Lift Process in L_7'

The following dictionary is used to code the input signals of the lift process on input tape S while simulating it by L_7' -program:

S - 1, R - 2, Z - 3, FU - 4, FD - 5, DOp - 6, DC - 7, TIM - 8.

Let us apply the transformation described in the previous subsection to the lift process. By this we note that signals FU and FD saved in state D-Opening can be retained in the input queue only while the process is in states Wait-Enter or Wait-Request. Due to the stated we build corresponding L_7' fragments for all other states as described in Step (2) of the transformation algorithm and define the variable t^4 which simulates delay time for signals FU and FD to be set to z just at the label D-Opening (hadn't we performed this optimization the resulting program would be a bit more complicated).

We also note that the value of t^8 in the simulating program can differ from (-1) only at the label WaitEnter, therefore the timer activity condition t^8=-1 will not be checked elsewhere.

So we obtain the program in L_7' depicted in fig. 24.1 thru 24.3.

10.5 Reachability Graph for Lift Program in L_7'

The reachability graph for the lift program in L_7' is built using the algorithm

described in the previous section, as well as some methods for reachability graph minimizing (similar to those described in Section 3 for L_0-programs).

We define essentially located statements (ELSs) to be the statements with labels attached to them except those with label "D-Opening" (this label is located "nearly at the same place" as "L2") and "L1".

In the construction of the reachability graph we use the following states (inequality systems) corresponding to program paths:

S1 = $\{-1 = t^8 < 0 = z, y = 1, Fl = 0\}$
S2 = $\{-1 = t^8 < 0 < z, y = 1, Fl = 0\}$
S3 = $\{-1 = t^8 < 0 < z = t^4, x = y = 1, Fl = 0\}$
S4 = $\{-1 = t^8 < 0 < t^4 < z, x = y = 1, Fl = 0\}$
S5 = $\{-1 < 0 < t^4 < z < z + 20 = t^8, x = y = 1, Fl = 0\}$
S6 = $\{-1 < 0 < z = t^4 < t^8 < z + 20, x = y = 1, Fl = 0\}$
S7 = $\{-1 < 0 < z = t^4 < z + 20 = t^8, x = y = 1, Fl = 0\}$
S8 = $\{-1 = t^8 < 0 < t^4 < z, x = y = 1, Fl = 1\}$
S9 = $\{-1 = t^8 < 0 < z = t^4, x = y = 1, Fl = 1\}$
S10 = $\{-1 = t^8 < 0 = z, x = y = 1, Fl = 0\}$
S11 = $\{-1 = t^8 < 0 = z, x > y = 1, Fl = 1\}$
S12 = $\{-1 = t^8 < 0 = z, x > y > 1, Fl = 1\}$
S13 = $\{-1 = t^8 < 0 < z = t^4, x = y > 1, Fl = 1\}$
S14 = $\{-1 = t^8 < 0 < t^4 < z, x = y > 1, Fl = 1\}$
S15 = $\{-1 < 0 < t^4 < z < z + 20 = t^8, x = y > 1, Fl = 0\}$
S16 = $\{-1 < 0 < z = t^4 < t^8 < z + 20, x = y > 1, Fl = 0\}$
S17 = $\{-1 < 0 < z = t^4 < z + 20 = t^8, x = y > 1, Fl = 0\}$
S18 = $\{-1 = t^8 < 0 = z, x = y > 1, Fl = 0\}$
S19 = $\{-1 = t^8 < 0 < z, y > 1, Fl = 0\}$
S20 = $\{-1 = t^8 < 0 < z = t^4, x = y > 1, Fl = 0\}$
S21 = $\{-1 = t^8 < 0 < t^4 < z, x = y > 1, Fl = 0\}$
S22 = $\{-1 = t^8 < 0 < z, x > y > 1, Fl = 0\}$
S23 = $\{-1 = t^8 < 0 = z, 0 < x < y, Fl = 0\}$
S24 = $\{-1 = t^8 < 0 = z, 0 < x = y, Fl = 0\}$
S25 = $\{-1 = t^8 < 0 = z, 0 < x < y, Fl = 1\}$
S26 = $\{-1 = t^8 < 0 = z, 0 < x = y, Fl = 1\}$
S27 = $\{-1 = t^8 < 0 < z, x > y = 1, Fl = 0\}$

Vertices of the reachability graph are pairs (ELS label, state corresponding to program path).

Let Li and Lj be labels of ELS's, and the path β from Li to Lj contains no other ELS's. There an edge corresponding to the path β is drawn from vertex (Li,Si) to (Lj,Sj) in the graph if $S(Si,\beta)=Sj$ (i.e., if the state Si is transformed into Sj by the path β according to inductive state building algorithm). The edge in the reachability graph corresponding to some path in the program will be labeled by exits of conditional statements defining the path (for the sake of brevity only exits of the statements with other exit not pending are shown in labels).

In order to make the representation of the reachability graph more compact and comprehensible we have chosen for every vertex the following kinds of paths:

1) from the vertex to stop,
2) from the vertex to itself,
3) not feasible

to be represented in special fields inside the image of the vertex. For example, the

vertex image

WaitRequest,S9		
-	+1,+2--	+4-,+2-+

represents the following fragment of graph:

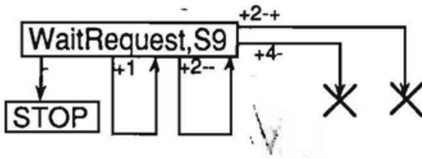


Fig.25

The constructed reachability graph is depicted in fig. 26.1 and 26.2.

During the construction of the graph a nondeterministic branching was admitted to reduce the size. Namely, while forming the inequality system S24 (S26 respectively), it is easy to see that the set of its solutions coincides with the union of solution sets for S3 and S20 (S13 and S9 respectively). Thus, instead of drawing an edge from (FlyingDown,S23) ((FlyingDown,S25) respectively) to (L2,S24) ((L2,S26) respectively) corresponding to path +3- we make a nondeterministic branching leading to both (L2,S3) and (L2,S20) ((L2,S13) and (L2,S10) respectively).

It is easy to see that nondeterministic branching causes no obstacles for finding the coverings of graph and solving corresponding inequality systems. If nondeterministic branching were not used, additional 11 states in the graph would have been necessary.

10.6 Path Inequality Systems: Example

Now let us show how to build an inequality system for some path in the lift program (being a projection of a path in the reachability graph) and find a test enforcing the execution of it.

Let us consider a path in the reachability graph v=(Idle,S1)+1+, (FlyingUp,S27)+3+, (FlyingUp,S22)+3-, (L2,S20)+6-, (WaitEnter,S15)+5+-, (WaiRequest,S14)+4-, (WaitEnter,S15)+1+, (WaitEnter,S16)+8, (DoorClosing,S18)+7-, (Idle,S19)-, STOP.

It has the following projection α in the program:

$(-1)\to t^8,\ 1\to y,\ T\to t+,\ t\dot{\to}z+,\ S\to s+,\ s{:}\ 1,\ P\dot{\to}x+,\ x>0+,\ x>y+,$

$T\to t+,\ t\dot{\to}z+,\ S\to s+,\ s{:}\ 3,\ P\to w+,\ w>y+,\ w\to y,\ w<x+,$

$T\to t+,\ t\dot{\to}z+,\ S\to s+,\ s{:}\ 3,\ P\to w+,\ w>y+,\ w\to y,\ w<x-,\ w>x-,$

$z\to t^4,\ T\to t+,\ t\dot{\to}z+,\ S\to s+,\ s{:}\ 6,\ Fl>0-,\ (z+20)\to t^8,$

$T\to t+,\ S\to s+,\ s{:}\ 5,\ t^8>-1+,\ t<t^8+,\ t\dot{\to}t^4+,\ t\dot{\to}z-,\ t<z+,\ (-1)\to t^8,\ 1\to Fl,$

$T\to t+,\ S\to s+,\ s{:}\ 4,\ t\dot{\to}t^4+,\ t\dot{\to}z-,\ t<z+,\ 1\to Fl,\ (z+20)\to t^8,$

$T\to t+,\ S\to s+,\ s{:}\ 1,\ t^8>-1+,\ t<t^8+,\ t\dot{\to}z+,\ t\dot{\to}t^4+,\ P\to u+,$

$T\to t+,\ S\to s+,\ s{:}\ 8,\ t\dot{\to}z+,\ t=t^8+,\ t\dot{\to}t^4+,\ (-1)\to t^8,$

$T\to t+,\ t\dot{\to}z+,\ S\to s+,\ s{:}\ 7,\ Fl>0-,$

$T\to t-,\ S\to s-,\ P\to u-,$ STOP.

There the following inequality system corresponds to the path α:

$z_0 = t_0 = t_0{}^4 = t_0{}^8 = 0,\ y_0 = x_0 = w_0 = Fl_0 = s_0 = 0;$

$t_1{}^8 = -1;\ y_1=1,\ t_1=T_1;\ t_1>z_0,\ z_1=t_1;\ s_1=S_1;\ s_1=1,\ x_1=P_1;\ x_1>0;\ x_1>y_1;$

$t_2=T_2;\ t_2>z_1,\ z_2=t_2;\ s_2=S_2;\ s_2=3;\ w_1=P_2;\ w_1>y_1;\ y_2=w_1;\ w_1<x_1;$

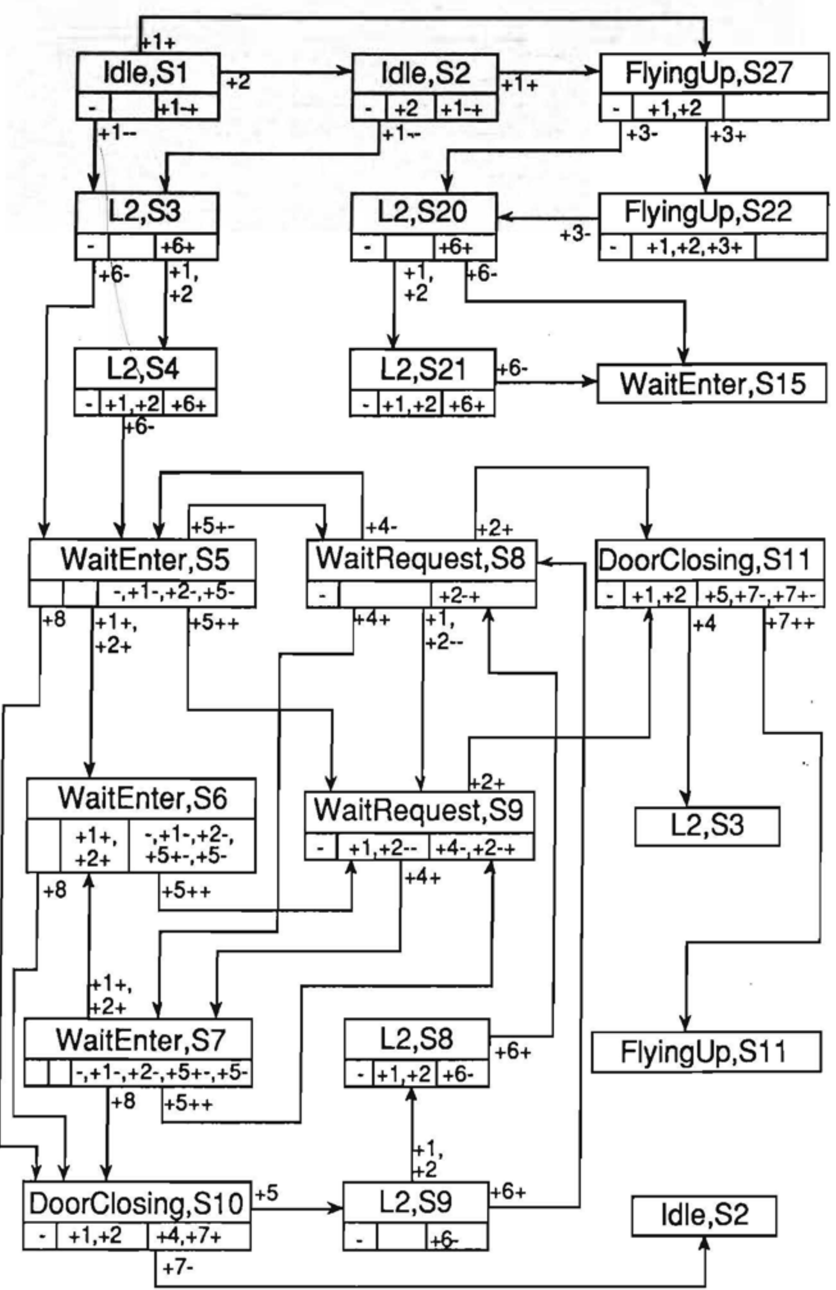


Fig.26.1

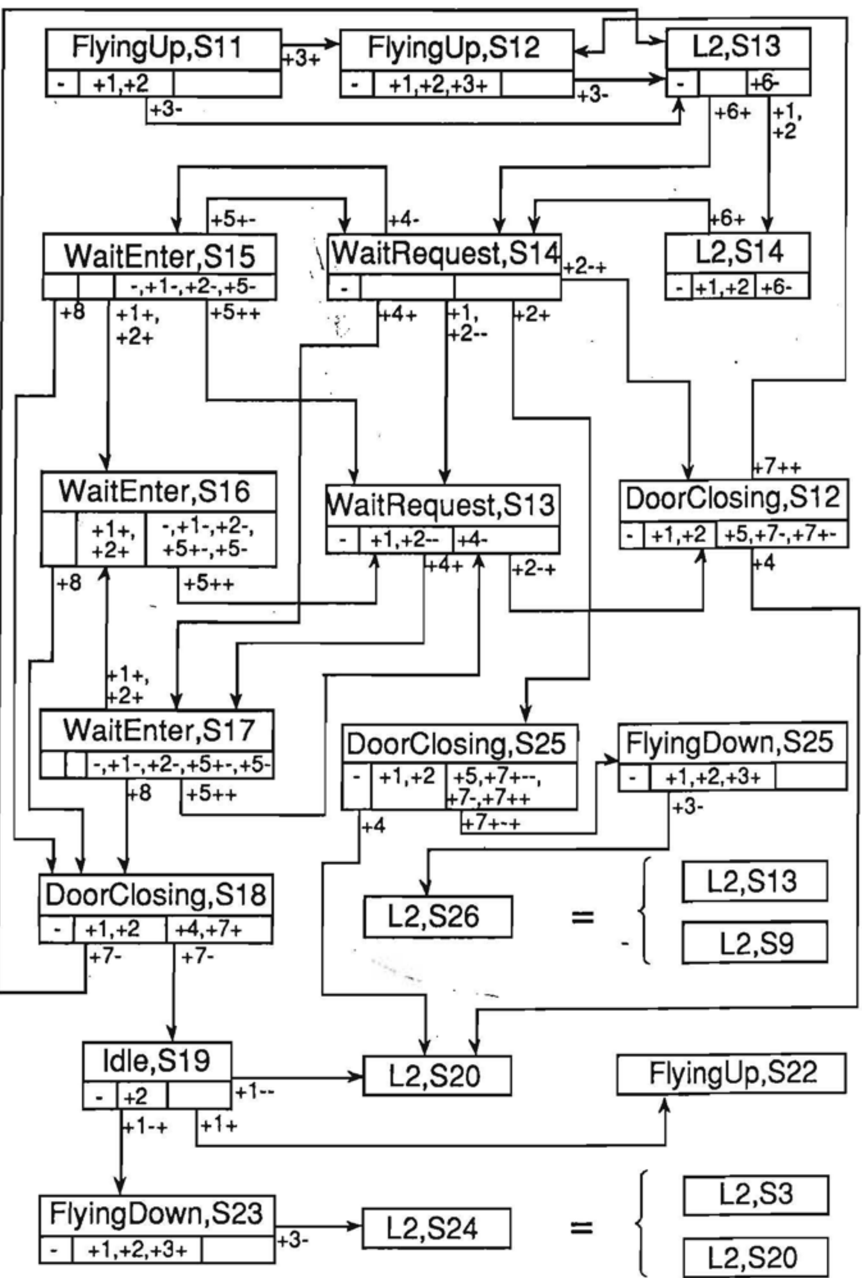


Fig.26.2

$t_3=T_3; t_3>z_2, z_3=t_3; s_3=S_3; s_3=3; w_2=P_3; w_2>y_2; y_3=w_2; w_2\geq x_1; w_2=x_1;$
$t_1^4=z_3; t_4=T_4; t_4>z_3, z_4=t_4; s_4=S_4; s_4=6; Fl_0\leq 0; t_1^8=z_4+20;$
$t_5=T_5; s_5=S_5; s_5=5; t_1^8>-1; t_5<t_1^8; t_5>t_1^4, t_2^4=t_5; t_5\leq z_4; t_5<z_4; t_2^8=-1; Fl_1=1;$
$t_6=T_6; s_6=S_6; s_6=4; t_6>t_2^4, t_3^4=t_6; t_6=z_4; t_6<z_4; Fl_2=0; t_3^8=z_4+20;$
$t_7=T_7; s_7=S_7; s_7=1; t_3^8=-1; t_7<t_3^8; t_7>z_4, z_5=t_7; t_7>t_3^4, t_4^4=t_7; u_1=P_4;$
$t_8=T_8; s_8=S_8; s_8=8; t_8>z_5, z_6=t_8; t_8=t_3^8; t_8>t_4^4, t_5^4=t_8; t_4^8=-1;$
$t_9=T_9; t_9>z_6, z_7=t_9; s_9=S_9; s_9=7; Fl_2\leq 0;$
$T<0; S<0; P<0.$

With respect to the values of input tape cells $T_1,..,T_9$, $S_1,..,S_9$, $P_1,..,P_4$ it is equivalent to the following inequality system:
$0<T_1<T_2<T_3<T_4<T_7<T_8<T_9;\ \ T_3<T_5<T_6<T_7<T_8;$
$T_5<T_4;\ \ T_6<T_4;\ \ -1<T_4+20;\ \ T_5<T_4+20;\ \ T_7<T_4+20=T_8;$
$S_1=1; S_2=3; S_3=3; S_4=6; S_5=5; S_6=4; S_7=1; S_8=8; S_9=7;$
$P_1>1; P_2>1; P_2<P_3; P_3>P_2; P_3=P_1.$

From this inequality system we can obtain, for example, the following test on which lift program traverses the path
T=(1,2,3,6,4,5,7,26,27) ; S=(1,3,3,6,5,4,1,8,7) ; P=(3,2,3,0)

Computational complexity of solving path inequality systems is not considered here, we note that the special form of path inequality systems is very essential for solving algorithm.

10.7 CTS for Lift Program

Using the constructed reachability graph for every branch in the program we can
1) determine whether it is feasible,
2) if so, find a feasible path containing the branch.

Further, by solving the inequality system for the obtained path, we find a test on which the given branch is executed.

Thus, considering consecutively all branches in the program, we construct a correct CTS for the program.

Choosing a definite order of branch consideration we obtain the following correct CTS for lift program.

<u>Test N1.</u> T=(1) , S=(2) , P=(0) ;

<u>Test N2.</u> T=(1,2,3,4) , S=(1,1,2,3) , P=(3,0,0,2) ;

<u>Test N3.</u> T=(1,2,3) , S=(1,1,2) , P=(1,0,0) ;

<u>Test N4.</u> T=(1,2,3,4,5,6,7,8) , S=(1,3,6,1,2,5,1,2) , P=(2,2,0,0,0,2) ;

<u>Test N5.</u> T=(1,3,2,4,5,6) , S=(1,6,5,2,1,2) , P=(1,2,0,0) ;

<u>Test N6.</u> T=(1,2,22,23) , S=(1,6,8,5) , P=(1) ;

<u>Test N7.</u> T=(1,2,3,4,5,6) , S=(1,3,6,5,2,4) , P=(2,2,1) ;

<u>Test N8.</u> T=(1,4,2,3,24,25,26,27,28) , S=(1,6,5,4,8,5,6,2,7) , P=(1,2) ;

<u>Test N9.</u> T=(1,2,3,4,5,6,7,8,9) , S=(1,3,6,5,2,7,1,2,3) , P=(3,3,1,0,0,2) ;

<u>Test N10.</u> T=(1,2,3,4,5,6,7) , S=(1,3,6,5,2,7,3) , P=(2,2,1,1) .

To conclude the analysis of lift example we demonstrate how to transform tests from the obtained CTS into tests for the lift SDL process (as it has been explained before, these tests will form correct CTS for the process according to analogue of criterion C_1).

Test N1. (R(0) at 1).
Test N2. (S(3) at 1), (S(0) at 2), (R(0) at 3), (Z(2) at 4).
Test N3. (S(1) at 1), (S(0) at 2), (R(0) at 3).
Test N4. (S(2) at 1), (Z(2) at 2), (DOp at 3), (S(0) at 4), (R(0) at 5), (FD at 6), (S(0) at 7), (R(2) at 8).
Test N5. (S(1) at 1), (FD at 2), (DOp at 3), (R(2) at 4), (S(0) at 5), (R(0) at 6).
(Let us note the different order of signals in the corresponding L'_7 test).
Test N6. (S(1) at 1), (DOp at 2), (FD at 3).
(Let us note that 4 signals were coded in the L'_7 test).
Test N7. (S(2) at 1), (Z(2) at 2), (DOp at 3), (FD at 4), (R(1) at 5), (FU at 6).
Test N8. (S(1) at 1), (FD at 2), (FU at 3), (DOp at 4), (FD at 25), (DOp at 26), (R(2) at 27), (DC at 28).
(See notes at tests N5 and N6).
Test N9. (S(3) at 1), (Z(3) at 2), (DOp at 3), (FD at 4), (R(1) at 5), (DC at 6), (S(0) at 7), (R(0) at 8), (Z(2) at 9).
Test N10. (S(2) at 1), (Z(2) at 2), (DOp at 3), (FD at 4), (R(1) at 5), (DC at 6), (Z(1) at 7).

11 Conclusions

In the mid 70-ies using the ideas described in Sections 2,3,4 an experimental CTS generation system for data processing programs (the system SMOTL [8,10]) was developed at the Computing Center of Latvia University. A COBOL–like language SMOD was used as source language for SMOTL. The system SMOTL was tested on many real business data processing programs. Experiments showed that SMOTL was able to build automatically complete test sets for the described class of programs at a speed comparable to that of high level language compilers. However, business data processing programs have no sufficiently high demands for their reliability to outweigh the additional efforts of developing and using automatic test generation systems. Therefore practical research in this direction was not continued.

The situation has changed essentially in the last few years when the necessity appeared to test complicated real time systems with very high demands on reliability. Automatic generation of test cases has sufficient practical importance for programs of this class. At the same time it is clear that automatic test generation is a very hard job for these systems. Theoretical foundation of test generation for systems of the kind is considered in Section 9. Practical methods for test generation are described in the companion paper [17].

REFERENCES

[1] D.S.Alberts. The economics of software quality assurance. In Proc. AFIPS Conf. 1976, pp. 433–442.
[2] A.I.Auzins. On the Construction of complete sample systems. Dokl. Akad. Nauk SSSR, Vol. 288, No. 3, 1984, pp. 564–568 (in Russian).
[3] A.I.Auzins. Decidability of the reachability for the relational push–down automata.

Programmirovanie, No. 3, 1984, pp. 3–12 (in Russian).
[4] J.M.Barzdin, J.J.Bicevskis, and A.A.Kalninsh. Construction of complete sample system for program testing. Latv. Gosudarst. Univ. Uch. Zapiski, Vol. 210, 1974, pp. 152–187 (in Russian).
[5] J.M.Barzdin, J.J.Bicevskis, and A.A.Kalninsh. Decidable and undecidable cases of the problem of Construction of the complete sample system. Latv. Gosudarst. Univ. Uch. Zapiski, Vol. 210, 1974, pp. 188–205 (in Russian).
[6] J.M.Barzdin, J.J.Bicevskis, and A.A.Kalninsh. Construction of complete sample system for correctness testing. Lecture Notes in Computer Science, Vol. 32, Springer–Verlag, 1975, pp. 1–12.
[7] J.M.Barzdin and A.A.Kalninsh. Construction of complete sample system for programs using direct access files. Latv. Gosudarst. Univ. Uch. Zapiski, Vol. 233, 1975, pp. 123–154 (in Russian).
[8] J.J.Bicevskis. Automatic construction of sample systems. Programmirovanie, No. 3, 1977, pp. 60–70 (in Russian).
[9] J.M.Barzdin, J.J.Bicevskis, and A.A.Kalninsh. Automatic construction of complete sample systems for program testing. In Proc. IFIP Congress, 1977, North–Holland, 1977, pp. 57–62.
[10] J.Bicevskis, J.Borzovs, U.Straujums, A.Zarins, and E.F.Miller. SMOTL–a system to construct samples for data processing program debugging. IEEE Transactions on Software Enqineering, SE–5, No. 1, 1979, pp. 60–66.
[11] E.F.Miller, Jr. Program testing technology in the 1980s. In Tutorial: Software Testing and Validation Techniques, 1978, pp. 399–406.
[12] M.L.Minsky. Finite and infinite machines. Prentice–Hall, Englewood Cliffs, N.Y., 1967.
[13] M.O.Rabin and D.Scott. Finite automata and their decision problems. IBM J. of Research and Development, vol. 3, No. 2, 1959, pp. 114–125.
[14] A.G.Tadevosjan. Decidable cases of the problem of construction of a complete sample system. Kibernetika, No. 6, 1985, pp. 41–44 (in Russian).
[15] K.C.Tai. Program testing complexity and test criteria. IEEE Trans. Software Engineering, SE–6, No. 6, 1980, pp. 531–538.
[16] CCITT Specification and Description Language (SDL), Recomendation Z. 100, 1988.
[17] J.Barzdins, J.Borzovs, A.Kalnins, I.Medvedis. Automatic construction of test sets: practical approach, this volume.

AUTOMATIC CONSTRUCTION OF TEST SETS: PRACTICAL APPROACH

Juris Borzovs, Audris Kalniņš, Inga Medvedis

Institute of Mathematics and Computer Science
The University of Latvia
Raiņa Bulv. 29, Riga 226250, Latvia

Abstract. The problem of symbolic execution and test generation is considered both for sequential and concurrent programs. Practical methods for test construction for the given program path are presented.

1. Introduction

Computer program testing (i.e., program execution on different input values - tests) remains an essential basis of program correctness decision. It is accepted that testing is not capable of program correctness proving (except cases when program is executed on all possible input values), nevertheless, in practice, if program gives correct outputs on sufficiently large amount of tests, confidence of its correctness becomes psychologically very strong.

As test generation is rather labor-consuming and quite often rather subjective, already tens of years ago trials were performed to automate this process [1,2]. One possible approach to the solution of the problem is test generation by means of symbolic execution of program paths and the following solution of path conditions (which mainly are systems of equalities and inequalities over program input parameters) obtained by symbolic execution.

Since 70-ies rather many experimental systems have been developed on the basis of the before mentioned approach [3-11]. In the second half of the 80-ies this approach has experienced the revival in the application of testing of specifications of large program systems (especially telecommunication) [12-14,28,29].

This paper deals with automated test generation methods for sequential programs and protocol specifications written in SDL language [15,16]. In both cases symbolic execution of programs is used.

The paper consists of two major parts. Part 2 deals with sequential programs. The notion of symbolic execution is formalized here. Sequential subset of SDL (equivalent to large part of Pascal) is described and an example of program is given. Correct symbolic

execution is defined for this subset of SDL and demonstrated on the program example. A heuristic method for solving equations (path conditions) obtained as the result of symbolic execution of program path is presented, thus yielding a practical method to generate a test executing a selected program path.

In Part 3 the approach is extended to concurrent programs. Correct symbolic execution is extended to all major concurrency concepts of SDL. The method is demonstrated on a realistic example - sliding window protocol in SDL, test generation procedure based on symbolic execution is shown for selected paths. Moreover, a heuristic method is presented for path selection (according to criterion C1) based on state concept related to that used for theoretical approach to test generation [17]. The method ensures the generation of test set executing all branches for the sliding window example in a reasonable time. A more sophisticated heuristic test generation method supposed to work efficiently on comparatively large SDL systems is also outlined.

Part 2 has been written by J.Borzovs and I.Medvedis. It contains results obtained by the authors at various times [10, 21, 22]. Part 3 has been written by A.Kalniņš and it contains new results.

2. Symbolic execution and test generation for sequential programs

Symbolic execution of programs is a wide area per se and has various applications. In this paper we restrict ourselves to the use of symbolic execution for feasibility condition description for program paths and test generation for a path based on these conditions.

Our approach to test generation by means of symbolic execution can be applied to a class of programming languages characterized by the following properties. These are block structured procedural languages with strong typing. A typical representative of this class is Pascal together with its newest derivatives like Modula-2, Turing, etc. Some restrictions, nevertheless, are present. We exclude direct memory management (pointers and related operations), calls of external procedures and functions as black boxes (with no source text available), nondeterministic functions (like random number generators).

Languages of the considered class have common property that the main control unit is a procedure with formal input parameters and declaration part defining local variables and their types. Procedure body consists of statements (assignments, conditionals, etc.) which can be represented both in conventional textual form and in flowchart like form.

Symbolic execution can be defined by our methods for any language of the class described. To do this a special symbolic execution language correlated to the given programming language must be designed. In this paper we describe only symbolic execution of SDL - in Part 2 for its sequential subset (equivalent in fact to large Pascal subset), in Part 3 we expand this definition to concurrent aspects of SDL.

2.1 Formalization of Symbolic Execution of Program Path

In order to define formally symbolic execution we must describe more precisely some notions present in any programming language L of the considered language class.

1. All data types permitted in the programming language L are denoted by T_1, T_2, T_3,.... If the language L permits only predefined types, then the number of the types used is finite. If the language L has type defining facilities (like **Array** [1..10] **Of** Integer in Pascal), then the number of possible types is infinite. Nevertheless we assume that syntax and semantics definition of the language L determines uniquely the complete type list T_1, T_2, ... and type declaration in a program is only a way to select one of these types. The domains of types (i.e., sets containing possible values of the variables of the type) are denoted by D_1, D_2, D_3, ... If the language L has type naming facilities like

```
Type Seqno = Integer;
     List  = Array[1..10] Of Seqno;
Var  Buf : List;
```

in Pascal, the corresponding <u>ground type</u> containing no intermediate program defined identifiers and having the same domain (**Array**[1..10] **Of** Integer for variable Buf in the example) is used to denote the type of variable in our discussions. A program independent list of possible ground types T_1,T_2,... can actually be defined for Pascal like languages (in a way similar to that used further for SDL subset).

2. We assume type T_1 to correspond to normal Boolean data type, so D_1 = {True,False}.

3. Every program P in the language L has a certain number n of variables. Each variable has a name and a certain type (from the list $T_1,T_2,\ldots$). If program P has variables $X_1,\ldots,X_n$, then the corresponding types are denoted by $T^P_{x_1},T^P_{x_2},\ldots T^P_{x_n}$ and domains by $D^P_{x_1},\ldots,D^P_{x_n}$.

4. A certain number of the program variables are input parameters, i.e., variables containing program input data. We assume the first m variables $X_1,\ldots,X_m$ to be the parameters.

5. The body of program P consists of statements, each of them having one or more exits (e.g., If-statement normally has two exits), all statements are somehow labelled. A sequence (S_1e_1, $S_2e_2,\ldots$, S_ke_k) is called a path if S_1 is the first statement of the program and if exit e_i of statement S_i determines S_{i+1} as the next statement to be executed. If statement S_i has only one exit or the exit is uniquely determined by some other syntactic means, e_i will not be given explicitly.

Now let us describe the symbolic execution of programs in the language L.

The <u>symbolic language</u> SL corresponding to the programming language L is defined as:

1. Many-sorted signature Σ defined over the same (ground) types $T_1,T_2,\ldots$ used in the programming language L. A special type T_0 with only one value undef in its domain D_0 is introduced (this value is used to denote undefined variable values). The signature contains function symbols $f_1,f_2,\ldots$ and for every function symbol f_i its argument types and result type are specified

$$f_i: T_{i_1}, \ldots, T_{i_k} \dashrightarrow T_{i_0}$$

(including zero argument constant functions) .No function f_i is defined over T_0, only constant function Undef has T_0 as value type.

2. An interpretation I of functions symbols $f_1,f_2,\ldots$ from signature Σ .

The main objects considered in the language SL are terms. Terms in SL are well-typed expressions composed of function symbols and typed variables in normal sense. Terms are used to describe the behaviour of programs in L. Though it is not required formally, function symbols $f_1,f_2,\ldots$ and their interpretation I normally isclosely related to functions used in the language L itself or in

its semantics description.

A term in SL is said to be a <u>predicate term</u> if its range is D_1= {True,False}.

We say that a term T <u>conforms</u> with a program P if all variables occurring in the term T are also input parameters of the program P and the type of the variable determined by its occurrence in the term T coincides with the type specified for the variable in the program P. If the language L uses type naming, then variable having some type in program P must have the corresponding ground type in term T (domains are the same!). Conformance informally means that term T is defined for the same entities which are processed by program P.

By <u>symbolic execution</u> (of programs in the language L) we understand an algorithm which, given a program P in the language L and a path α in P, produces:

1) a predicate term PC conforming with the program P,

2) for each variable $x_i (i=1,\dots,n)$ of program P a term T_{x_i} conforming with program P such that range of T_{x_i} according to signature $\sum$ coincides with the value range of x_i determined by its type (or the range is D_0={Undef}).

The predicate term PC is called <u>path condition</u>, the terms $T_{x_1},\dots,T_{x_n}$ associated with variables - <u>system of symbolic values</u>, and both of them together <u>symbolic state</u>.

Symbolic execution is said to be <u>correct</u> if it produces for every program P in the language L and for every path α in P a symbolic state such that, for all parameter values of program P $a_1 \in D^P_{x_1}$, $a_2 \in D^P_{x_2}, \dots,$ $a_m \in D^P_{x_m}$, there holds:

1) path condition $PC(a_1,a_2,\dots,a_m)$=true iff the program P executed on parameter values $a_1,\dots,a_m$ traverses the path α,

2) if term T_{x_i} is associated with variable $x_i (i=1,\dots, n)$ according to the system of symbolic values and the value of the instantiated term $T_{x_i}(a_1,\dots,a_m)$ is z_{x_i}, then after the program P has traversed the path α on parameter values $a_1,a_2,\dots,a_m$, variable x_i contains the same value z_{x_i}; if T_{x_i} = Undef, then variable x_i has no value assigned on path α.

So informally path condition is an assertion on parameter values of program P in order to force the execution of this path. Path condition actually accumulates the information from the

conditional statements (If, Case,...) traversed in the path . Some assertions to prevent from overflow-like errors are also accumulated in path condition. In test generation applications path condition is used to find parameter values (i.e., a test) forcing the execution of the path.

To summarize the above mentioned we can say that symbolic execution definition for some programming language L requires three tasks to be done:

1. symbolic language corresponding to L must be defined,
2. symbolic execution algorithm must be constructed,
3. correctness of symbolic execution must be proved.

In practice the majority ofmost attention usually is paid to symbolic execution algorithm, nevertheless, the two other items are important as well.

The next sections are devoted to symbolic execution definition for a certain programming language.

2.2 Programing Language

In this section we consider a simple sequential programming language. Constructions of the language we denote in traditional SDL [15,16] graphical form, adhering completely to SDL syntax, although many typical SDL language constructions (such as state, signal, timer...) do not occur. The considered subset of SDL functionally do not exceed the capability of Pascal language, therefore, in order to improve readability, sometimes we present translation of SDL construction in Pascal terminology.

This simple sequential programming language is used to demonstrate symbolic execution and test generation algorithms later on. We stress that the scope of the language constructions could be substantially wider from the point of view of our methods. However, we shall consider only those language constructions having been used in examples.

In this part test generation methods are demonstrated on separate SDL procedure. In SDL language the procedure has textual and graphic parts. Textual part contains the description of procedure formal parameters, types and variables. Graphic part describes data manipulations and control flow. Further we describe this language more precisely.

Data type definitions

1. Our language has three predefined data types: Integer, Boolean and Real associated with usual operations:

```
Newtype Integer
     Literals 0,1,2,3,... ;
     Operators
          "+"   : Integer, Integer -> Integer ;
          "-"   : Integer, Integer -> Integer ;
          "mod" : Integer, Integer -> Integer ;
          "="   : Integer, Integer -> Boolean ;
          "/="  : Integer, Integer -> Boolean ;
          "<"   : Integer, Integer -> Boolean ;
          ">"   : Integer, Integer -> Boolean ;
          "<="  : Integer, Integer -> Boolean ;
          ">="  : Integer, Integer -> Boolean ;
Endnewtype Integer;
Newtype Boolean
     Literals True, False;
     Operators
          "NOT" : Boolean -> Boolean ;
          "AND" : Boolean, Boolean -> Boolean ;
          "OR"  : Boolean, Boolean -> Boolean ;
          "="   : Boolean, Boolean -> Boolean ;
          "/="  : Boolean, Boolean -> Boolean ;
Endnewtype Boolean ;
Newtype Real
     Literals ...
     Operators
          "+"   : Real, Real -> Real ;
          "-"   : Real, Real -> Real ;
          "="   : Real, Real -> Boolean ;
          "/="  : Real, Real -> Boolean ;
          "<"   : Real, Real -> Boolean ;
          ">"   : Real, Real -> Boolean ;
          "<="  : Real, Real -> Boolean ;
          ">="  : Real, Real -> Boolean ;
Endnewtype Real
```

2. Subranges of Integer type with the following declaration:

```
Syntype Mytype=Integer
     Constants First : Last ;
Endsyntype Mytype ;
```

According to SDL semantics the behaviour of the subrange type is the same as the behaviour of the Integer type with the only difference that during the assignment of a value to subrange type variable (and in some other special cases) the range check of the value is performed.

3. Enumerated type with the following declaration:

```
Newtype Mytype
     Literals Lit1, Lit2, Lit3 ... ;
Endnewtype Mytype ;
```

For these types only the following equality relations are defined:

```
"="   : Mytype, Mytype -> Boolean ;
"/="  : Mytype, Mytype -> Boolean ;
```

4. Records (structs) with fields of any type mentioned above:

```
Newtype Mytype
     Struct
        Field1  Type1;
        Field2  Type2; ...
Endnewtype Mytype
```

5. Arrays with integer subscripts and values of any type mentioned above (including structs):

```
Newtype Mytype
     Array (Type1, Type2)
Endnewtype Mytype;
```

Here Type1 - type of index, Type2 - type of value.

Ground types corresponding to the introduced SDL types will be described in Section 2.3.

Statements of textual part

Along with type declarations textual part may also contain the following statements.

1. Procedure heading which is the first statement in every procedure specifying its name and describing its formal parameters:

```
Procedure Myproc
     Fpar
        In      Parameter1  Type1, ...
        In/Out  Parameter2  Type2, ...
```

Types Type1, Type2... must be defined outside the procedure or must be predefined.

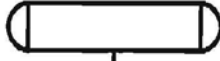
2. Declarations of variables:

```
     DCL
        Var1  Type1,
        Var2  Type2 ... ;
```

In the case of embedded procedures usual visibility rules are valid.

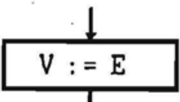
Statements of graphic part

1. Procedure start:



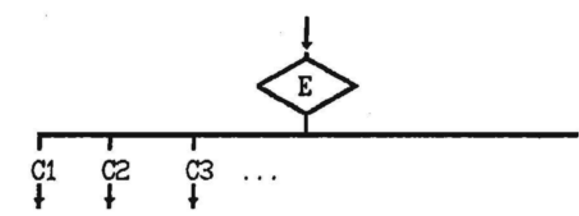
2. Procedure termination:

3. Assignment statement:



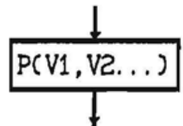
Here V is either name of variable, element of array A(I), structure element S!F or element of array of structures S(I)!F. Symbol E denotes expression of appropriate type in the usual sense.

4. Decision statement:



where E is expression of scalar type (except Real) and C1,C2,C3 ... - constants of the same type.

5. Procedure call:



where P - name of procedure; V1, V2 ... - variables or expressions of appropriate type.

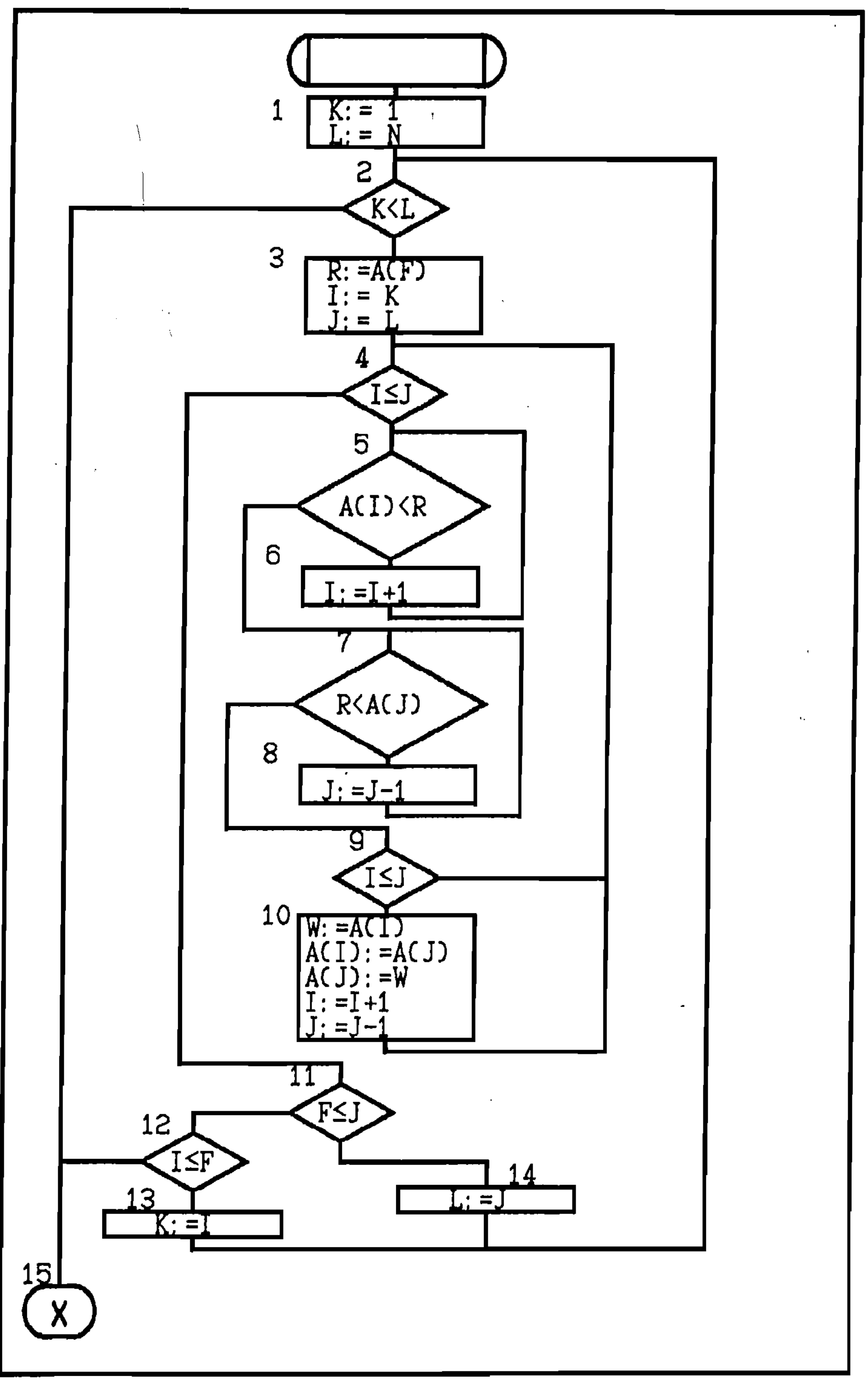
Example of Sequential Program

Let us consider a program FIND [19] which is often used to demonstrate different techniques of verification and testing.

Input of the program FIND is integer array A, its length N and some integer F. The purpose of the program is to find the element of array A whose value is F-th in the order of magnitude and to rearrange the array in such a way that this element is placed in A(F) and, furthermore, all elements with subscripts lower than F have lesser values and all elements with subscripts greater than F have greater values. Thus on completion of the program the following relationship holds:

A(1), A(2), ... A(F-1) <= A(F) <= A(F+1), ... A(N)

```
Syntype Int=Integer
     Constants 1:100;
Endsyntype Int;
Newtype IA
     Array(Int,Int)
Endnewtype IA;

Procedure Find
     Fpar
        In/Out A   IA,
        In     N   Int,
               F   Int;

     Dcl
               K   Int,
               L   Int,
               I   Int,
               J   Int,
               R   Int,
               W   Int;
```


1
K: = 1
L: = N
2
K<L
3
R: =A(F)
I: = K
J: = L
4
I≤J
5
A(I)<R
6
I: =I+1
7
R<A(J)
8
J: =J-1
9
I≤J
10
W: =A(I)
A(I): =A(J)
A(J): =W
I: =I+1
J: =J-1
11
F≤J
12
I≤F
13
K: =I
14
L: =J
15
X

2.3 Symbolic Language

Here we describe a symbolic language SDLS corresponding to our SDL subset. Let us remind that we have to define many-sorted signature and its interpretation.

However, at first we must present the list of ground types for our SDL subset. Since SDL has both type defining and type naming facilities the concept of ground type is non-trivial here. So we consider all type defining constructs in SDL subset and introduce notations for the corresponding ground types.

1. Predefined types. The three predefined types: Boolean, Integer and Real are defined as ground types for all uses of these types under synonym names. For example, if type declarations

```
Syntype Counter = Integer
Endsyntype Counter;
Syntype Index = Counter
Endsyntype Index;
```

are used, the ground type corresponding to Index type is Integer.

2. All Integer subranges have the same ground type - Integer. The treatment of range checking is discussed later.

3. Enumerated types. We assume that ground type corresponding to an enumerated type determines only the number of constants, not their names. Notations for ground types are "Enumerated 1" for enumerated types with one constant, "Enumerated 2" for enumerated types with two constants, etc. For example, if we have

```
Newtype Color
     Literals Black, White ;
Endnewtype Color;
Newtype Sex
     Literals Male, Female ;
Endnewtype Sex;
```

then ground type Enumerated 2 correspond to both type Sex and type Color (Appropriate adjustment of constant functions see later).

4. Records (structs). Ground type for structs is defined as a list of ground types of struct fields. For example, if we have

```
Newtype Sex
     Literals Male, Female ;
Endnewtype Sex;
Newtype Person
     Struct
```

```
                    Sex_of_Person  Sex;
                    Age_of_Person  Integer ;
          Endnewtype Person;
```

then the ground type corresponding to type Person is Struct (Enumerated 2, Integer).

5. Arrays. Ground type for arrays defines the range of index and ground type of components. For example, if we have

```
          Syntype Index=Integer
               Constants 1:100 ;
          Endsyntype Index;
          Newtype Table
               Array(Index, Real)
          Endnewtype Table;
```

then the ground type of Table is Array(1:100, Real).

This completes the list of ground types (cf. $T_1, T_2, \ldots$ in 2.1). Domains corresponding to ground types follow straightforwardly from the language definition so they are discussed no more.

Now let us introduce function symbols of SDLS signature. The interpretation is described only for those functions where it is not obvious. Function symbols mainly are based on the operators introduced in our programming language (+, -, =, /=,...), and they use the same infix notation. However, most of the operators in our language are overloaded (i.e., defined for various data types simultaneously). Therefore SDLS contains derived function symbols for each of the ground types. So we have functions +, -, mod, =, /=... for Integer type; $\neg$, &, $\vee$, $=_{Boolean}$, $/=_{Boolean}$ for Boolean type and $+_{Real}$, $-_{Real}$, $=_{Real}$, $/=_{Real}$,... for Real type.

Four new groups of function symbols are introduced for complex data types (arrays and structs). They are based on functions used in SDL semantics definition.

1. Functions of extraction of value of array element $extract_T(a,i)$, where T - ground type of array as described above. These functions have array as the first argument and array index as the second one. Function yields value of corresponding array element.

2. Functions of modification of array $modify_T(a,i,v)$, where T - ground type of array, a - array, i- index of element and v- new value of element. Function yields a new array differing from the array a in the modified element i.

3. Functions of extraction of value of structure element

$extract_T(s,i)$. Functions yield the i-th element in structure s.

4. Functions of modification of structure element $modify_T(s,i,v)$. Functions yield a new structure with the value of the i-th element set to v.

In order to describe array operations adequately we assume that the first argument (array) of the function $modify_T$ can assume both values from the domain determined by type T and special value Undef of type T_0. (We could be completely formal in this case and introduce an auxiliary function $Undefarray_T$: T_0--> T, but this would make array expressions more awkward). Thus the interpretation of $modify_T$ is extended in a natural way, so that, e.g., the term $extract_{Array(1:10,Integer)}(Modify_{Array(1:10,Integer)}(Undef,1,3),1)$, has value 3.

The domain of modify function for structs is extended in a similar way.

The signature of SDLS also contain functions with zero arguments or constants. The same notations for constants as in SDL are used (of course, excluding overloading by means of ground type postfixes). So, e.g., the signature contains constants 1,2,3,... for Integer type, True and False for Boolean type and $1_{Real}, 2_{Real}, 3_{Real}, \ldots$ for Real type. New constant notations are introduced for enumerated types (in accordance with the corresponding ground type definitions). So the type Enumerated 2 has two constants $1_{Enumerated2}$, $2_{Enumerated\ 2}$. Constant notations adopted in SDL are used for arrays and structs, for example, $(.1,2,3.)_{Array(1:3,Integer)}$ or $(.1,18.)_{Struct(Enumerated2,Integer)}$.

This concludes the definition of the symbolic language SDLS. Unfortunately, terms in this language look very lengthy. For example, if we use data types from the program FIND, a correct term would be $extract_{Array(1:100,Integer)}(A,F)$. To make terms more readable we introduce a new notation system called the <u>derived</u> symbolic language. In this language as function and constant postfixes we do not use ground type notations but corresponding type names from the program declarations. So the beforementioned term in the derived symbolic language is $extract_{IA}(A,F)$. This is no more a correct term in the signature of SDLS, since there is no function $extract_{IA}$ in it and it can't be introduced unambiguously because identifier IA can designate various types in different programs. However, if we consider pairs <type declarations in program, term in derived language>, evidently there is an algorithm yielding an

equivalent term in SDLS for such a pair. For this reason we use the derived language without any special indication.

2.4 Algorithm of Symbolic Execution of Program Path

The aim of symbolic execution of program path is to obtain correct symbolic state.

Assume that a program path is given. In the case of procedure call program path contains also the corresponding sequence of statements of the called procedure. We shall show how to build symbolic state traversing the given path statement by statement.

Procedure Start

If symbolic execution begins with the given procedure, then variables - input parameters are assigned terms consisting of single variable, namely, the parameter itself. The rest of procedure variables are assigned undefined values (i.e., term Undef). The path condition is assigned term True.

If, on the contrary, we have reached this statement from other procedure, then, according to the range of accessibility of variable names, those pairs of variables are determined whose values are the same in the caller procedure and in this one. Formal parameters are assigned the same symbolic values as actual parameters in the caller procedure have (or terms formed by call statement). Local variables are assigned Undef values.

In both cases, if some of the input parameters are of subrange type, we also add (by means of & function) appropriate range checking predicate (such as N >= 1 & N <= 100) to the path condition.

Assignment Statement

The execution of assignment statement consists of value extraction of expression operands, calculation of value of expression and, the last, assignment of calculated value to the variable located in the lefthand side of the statement.

1. Extraction of values.

Due to the correspondence of names and values within the system

of symbolic values of current symbolic state we find symbolic values of variables contained in the righthand side of the statement.

If the considered variable is an element of array or record (struct), then we must form a new term using functional symbols of extract type. We also add predicates to the path condition to ensure that the range of indexes for the array is not violated.

For constants contained in the righthand side their ground types are determined (according to SDL typing rules for expressions) and the corresponding constant denotations are found.

2. Calculation of value of expression.

Taking previously obtained operand terms and functional symbols associated with corresponding operations (with correct type postfixes found) we construct a new term. If any of the operands has Undef as symbolic value, the resulting term is also Undef.

3. Assignment of value.

If the lefthand side of the statement contains scalar variable (or whole array), then the latter is assigned the newly obtained term in the system of symbolic values. If the variable is of subrange type, we add range checking predicate to the path condition, too.

If, on the contrary, the lefthand side contains array or record element, then the term of the modified value is constructed by means of modify function and this new value is assigned to the corresponding variable (i.e., array or record) in the system of symbolic values. We also enhance the path condition to ensure that the range of indexes is not violated.

Procedure Call

Here we remind that program path contains also corresponding sequences of statements of called procedures, and we permit only calls of procedures whose texts are available.

Therefore the next statement of the path is procedure start statement of the called procedure, and the given statement will be executed when we determine the binding of actual and formal parameters of procedure statement. If actual parameter is an expression, the corresponding term is formed as in assignment statement.

Decision Statement

As in the case of assignment statement we extract values of operands (terms) and construct the resulting term. If its type is Boolean, then such decision statement is called **If** statement, otherwise, Case statement.

In case of **If**, if the path proceeds along True-exit, the newly constructed term (simultaneously it is also a predicate term) is added to the path condition by means of & function. If the path proceeds along False-exit, we form negation of the previous term by means of ¬ function and add it to the path condition.

In the case of **Case** a new predicate term is constructed by the help of function = (equality) with the above mentioned term as the first argument, but the second argument is the constant assigned to the corresponding exit of the statement in the program. This last predicate term is added to the path condition.

Procedure Exit

Local variables of the procedure are removed from the system of symbolic values.

This concludes the definition of symbolic execution. It remains to formulate the following assertion:

The symbolic execution defined in this section is correct for SDL subset described in Section 2.2.

The proof of this assertion is a little bit lengthy and is left to very patient readers.

We just note that formally correctness refers only to the basic form of symbolic language. As far as this form can be uniquely restored from the derived form and program declarations the derived form is also correct in some sense and henceforth only this form is used (sometimes omitting type qualifiers for owerloaded functions at all, if they can be uniquely restored from the context).

Simplifier of Symbolic State

Whenever a new term is assigned to any variable or a new predicate is added to path condition we try to simplify this symbolic value or path condition. Our simplifier is rather primitive

and is mainly designed to find and calculate constant subterms in expressions. The simplifier is able to perform the following transformations:

1. Find out and calculate numerical and enumerated subterms composed of constants. Term ((1+x)+1)+1, for example, is reduced to x+3.

2. Find out and calculate subterms composed of array-type constants. Simplify array-type terms according to the following rules:

$extract_T(modify_T(A,i,x),i) \rightarrow x$
$modify_T(modify_T(A,i,x),i,y) \rightarrow modify_T(A,i,y)$
If i /= j then $extract_T(modify_T(A,i,x),j) \rightarrow extract_T(A,j)$

. . .

Here T - type of array; A - array-type term; i,j,x,y - scalar terms.

3. Simplify record-type terms the same way as arrays.

4. Reduce predicate terms to normal form. We define the normal form as conjunction P_1 & P_2 & ... P_n. Here P_i - elementary relations in form E op F , where E and F are numerical (enumerated) type terms and op is one of the operations =, /=, >, ... Our normal form is a special case of the conjunctive normal form and, of course, arbitrary predicate term can't be reduced to such a form. Nevertheless, in order to simplify material presentation, we discuss predicate terms only in normal form.

5. Simplify elementary relations (i.e., E > F & E < F -> False).

6. Calculate constant predicate terms (i.e., P & False -> False).

It should be noted that in this section we give only examples of simplification rules, not a complete list of them.

2.5 Example of Symbolic Execution of Program Path

Here we do not analyze particular methods of program path selection, although one of them actually is used to select program paths to be executed (the fundamental principle is to proceed along those feasible branches having been selected less frequently; in the case of several equal variants a generator of pseudo random numbers is used).

We apply symbolic execution to the program FIND mentioned in Section 2.2, namely, to path :

(1,2,3,4,5,6,5,7,9,10,4,5,7,8,7,9,4,11,14,2,3,4,5,6,5,7, 9,10,4,11,12,15).

After symbolic execution of procedure start statement the symbolic state is as follows:

System of symbolic values	Path condition
A = A	N >= 1 & N <= 100 &
N = N	F >= 1 & F <= 100 &
F = F	$extract_{IA}$(A,1) >= 1 &
K = undef	$extract_{IA}$(A,1) <= 100 &
L = undef	. . .
I = undef	$extract_{IA}$(A,100) >= 1 &
J = undef	$extract_{IA}$(A,100) <= 100
R = undef	
W = undef	

Parameters of the procedure are assigned terms consisting of single term variable, all other program variables are assigned undef term and the path condition consists of range checking predicates. Further for the sake of brevity we demonstrate only changes of symbolic state. If some term can be simplified by our simplifier, we show the result of the simplification (especially it refers to the range checking predicates).

After statement 1 (K:=1; L:=N) the symbolic state is changed as follows:

System of symbolic values	Path condition
K = 1	No changes
L = N	

After statement 2 (If K<L true exit):

No changes	1 < N

After statement 3 (R:=A(F); I:=K; J:=L):

R = $extract_{IA}$(A,F)	No changes
I = 1	
J = N	

After statement 4 (I<=J true exit):

no changes

no changes (1 <= N is reduced by simplifier)

We conclude the example by showing the symbolic state at the end of the given path. We use the following shorthand denotation:

$B = modify_{IA}(modify_{IA}(A,2,extract_{IA}(A,N)),N,extract_{IA}(A,2))$

The resulting symbolic state is as follows:

System of symbolic values

$A = modify_{IA}(modify_{IA}(B,2,extract_{IA}(B,N-2)), N-2,extract_{IA}(B,2))$

N = N

F = F

K = 1

L = N-2

I = 3

J = N-3

$R = extract_{IA}(B,F)$

$W = extract_{IA}(B,2)$

Path condition

$extract_{IA}(A,1) < extract_{IA}(A,F)$ &

$extract_{IA}(A,2) >= extract_{IA}(A,F)$ &

$extract_{IA}(A,F) >= extract_{IA}(A,N)$ &

4 <= N &

$extract_{IA}(A,3) >= extract_{IA}(B,F)$ &

$extract_{IA}(B,F) < extract_{IA}(B,N-1)$ &

$extract_{IA}(B,F) >= extract_{IA}(B,N-1)$ &

5 > N &

F <= N-2 &

$extract_{IA}(A,1) < extract_{IA}(B,F)$ &

$extract_{IA}(A,N) >= extract_{IA}(B,F)$ &

$extract_{IA}(B,F) >= extract_{IA}(B,N-2)$ &

F > N - 3 &

3 > F

2.6 Method for Solving Path Conditions

In the result of the symbolic execution of program path we obtain path condition $PC(x_1 \dots x_n)$, where x_i - scalar, array or record type variable. In order to find a test case which forces this path to be executed we must solve PC as a system of equalities (inequalities). The fact that PC is reduced to normal form is irrelevant for our solution algorithm; it is used only to simplify the explanation.

Before we begin solving the path condition we, first, free it from variables and functions of record type. It can be done easily because we can assume that record fields are independent variables or arrays (if array of records). Next we separate the given path condition into independent components $PC(x_1 \dots x_n) = P_1(x_1 \dots x_i)$ & $P_2(x_{i+1} \dots x_j)$ & ..., where P_k and P_l have no common variables. After that we begin to solve these independent components.

Our method (see method of segments in [22]) is in fact exhaustive search algorithm which is improved by a number of heuristics. These heuristics are based on the study of real-life programs and are proved to be useful in test generation systems [10,22].

First let us sketch pure exhaustive search algorithm:

```
1 Procedure Resolve(P:Path_condition);
2     Select X - any variable or element of array in P;
3     For C := all possible values of X do
4         Q := P with X fixed to C;
5         Simplify Q;
6         If Q = True
7            Then System solved;
8         If Q /= True & Q /= False
9            Then Resolve(Q);
10    End
11 End Resolve;
```

To fix the value of X to C in step 4 we simply replace X by constant C or, in the case when X is an element of array (i.e., A(I)), we replace A by modify(A,I,X). In step 5 the above described simplification procedure is used to determine how successful our fixations were.

This algorithm, of course, can solve any path condition, but it is extremely impractical. After the improvements the algorithm becomes much faster, but it is not able to solve some very complex path conditions. Nevertheless, inability to solve some path condition is not dangerous for test generation system. It may lead (and even then not always) only to test systems with lower quality.

We discuss heuristics related to the three steps of the given algorithm.

First, in step 2 we select the next variable to be fixed. In practice the sequence in which we fix variables is very important for the speed-up of algorithm [8].

Second, in step 3 we try all possible values of the given variable. Yet, most of these values are not useful a priori [10].

Third, when values are fixed in step 3, first of all we must try those values that are more likely to be solutions of path condition.

Let us discuss these three heuristics separately.

Selection of Next Variable

The following criteria are used to select the next variable to be fixed:

1. Select only scalar variables or elements of arrays that are addressed in path condition with constant index (i.e., if path condition contains extract(A,5), then we are allowed to fix A(5)). It is easy to see that this criterion can never lead us to a situation when none of the variables can be selected.

2. If criterion 1 leaves some freedom for selection, we find in path condition the elementary relation containing the least number of variables and we fix one of these variables (according to criterion 1). It allows us to simplify the path condition as early as possible.

3. If criterion 2 also leaves some freedom, we select a variable with the smallest set of admissible values (see below).

Set of Admissible Values

With every variable in path condition we associate a set of admissible values, namely, some segment [a,b], such that no value

outside the segment can act as solution of the given path condition. We do not worry if some value inside the segment can never be solution of the system, but we are interested to keep these segments as small as possible.

For Boolean and Enumerated variables we discuss only two types of sets of admissible values: "any value (no limitations)" and "only one value C admitted". Further these sets of admissible values we call segments.

Before we begin to solve the path condition we find initial segments of variables. After fixation of every new value we revise them. First, we can set up initial segments according to the declaration of variable (for example, if the variable is a subrange). Second, a very valuable source of information is the user of the test generation system. Single remark, such as: "I am interested only in the case when all variables are less than 10", can significantly improve the performance. Third, the source of initial segments can be input/output formats, the use of variable in some language constructs, etc.

The most interesting procedure of our algorithm is reduction of segments with respect to the path condition. The aim of this procedure is to make our segments as small as possible. We apply this procedure any time when a new value of variable is fixed (after step 5). For example, if the current path condition is $x+y<z \ \& \ x>2$ and all variables are of integer type with equal segments [1,9], we are able to reduce the segment of x to [3,7], the segment of y to [1,5] and the segment of z to [5,9]. Reduction of segments is based on simple properties of arithmetic operations and relations. For example: "If the segment of x is [a1,a2], the segment of y is [b1,b2] and the value of x+y must be in segment [c1,c2], then the segment of x can be reduced to [max(a1,c1-b1), min(a2,c2-b1)], the segment of y can be reduced to [max(b1,c1-a2),min(b2,c2-a1)] but the value of x+y must be in the segment [max(c1,a1+b1),min(c2,a2+b2)]". Or another example: "If the segment of x is [a1,a2] and path condition contains relation x=b, then the segment of x can be reduced to [b,b]".

With iterative application of these local reductions we can propagate improvements through entire path condition. The algorithm of propagation is not quite trivial and includes some new heuristics for performance improvement, yet we do not discuss it in detail.

Selection of Values

It is possible that even after segment reduction exhaustive search is not useful. For that reason we first try to fix some outstanding values of variable: 1) both ends of segment, 2) those values within the segment that appear in the program text as constants, 3) one arbitrary point between every two values mentioned above. These rules (like our algorithm as a whole) are very simple but they work on real-life programs.

Example of Test Generation

Now we demonstrate how the path condition produced at the end of the previous section can be solved by our methods. It is easy to see that this path condition is only roughly simplified but we do not need a stronger simplifier because our method of segments can take into account all relations between variables.

We begin with the setup of initial segments. According to the declarations, segments of all variables (i.e., N, F and A) are set to [1,100]. The second step is the reduction of segments. During reduction the segment of N is improved to [4,4], the segment of F to [2,2] but segments of A are not significantly improved. Now we must fix a value of one of the variables. It was suggested that scalar variables must be fixed first, so we can fix N to 4 (no choice).

Next we perform the simplification and the reduction of segments once more. Then, the same way, we fix F to 2 and after just another simplification and reduction of segments get the following results:

Path condition:

$extract_{IA}(A,1) < extract_{IA}(A,2)$

$extract_{IA}(A,1) < extract_{IA}(A,4)$

$extract_{IA}(A,2) >= extract_{IA}(A,4)$

$extract_{IA}(A,3) > extract_{IA}(A,4)$

Segments:

$extract_{IA}(A,1)$	[1,98]
$extract_{IA}(A,2)$	[2,100]
$extract_{IA}(A,3)$	[3,100]
$extract_{IA}(A,4)$	[2,99]

Now one element of A is to be fixed. All elements are accessed with constant indexes and three of them have segments of equal size (i.e., 1-st, 3-rd and 4-th). Let us assume that we select the 3-rd element at random and fix it to 3 (the left end of the segment). This time the following reduction of segments is not trivial, nevertheless, it is very successful:

$extract_{IA}(A,1)$ [1,1]
$extract_{IA}(A,2)$ [2,100]
$extract_{IA}(A,4)$ [2,2]

So we proceed until the system is solved. It should be noted that during the solution of this system we are never forced to step back and fix a variable repeatedly. The resulting test is as follows:

A = (.1,2,3,2.)
N = 4
F = 2

If one tries to build a test for the same path manually, he probably will get a slightly different array A = (.1,4,3,2.) and will expect the procedure FIND to exchange the 2-nd and the 4-th elements. The test we have built is not so natural but it shows a significant drawback of the procedure FIND - although input array has been already partially sorted the procedure wastes time to exchange equal elements of the array.

Our path selection method coupled with the above mentioned test generation algorithm produce the following set of tests:

A = (.1,2,3,2.)	A = (.1.)	A = (.1,3,2,4.)
N = 4	N = 1	N = 4
F = 2	F = 1	F = 2

2.7 Abstract Data Types and Symbolic Execution

So far we have considered only programs with predefined data types. As we know, when new data types are introduced in SDL, their semantics is specified by means of axioms. So let some new types $t_1,t_2,...$ with new operators $o_1,o_2,...$ of some fixed signatures be

given. The new operators are just included in the definition of symbolic functional term. (Now the formal definition of symbolic execution language changes for program to program even for the basic form of the language). The main problem is how to cope with a widened class of terms while simplifying symbolic values and solving path conditions. So we request axioms for new types and operators to be respecified as term rewriting system (TRS) rules [23,27]. The TRS should be as good as possible - confluent and terminating.

TRS describing the new types is supplied to the simplifier. As we see from the general description of the simplifier (2.4), its basic action (simplifying arithmetic, logic, array and struct terms) could also be in fact described by means of TRS (though not always with a unique normal form, due to commutative rules). So the new and basic rules are merged together making a single TRS for both old and new types. So the simplifier tries to simplify any symbolic value of a variable using this TRS as far as possible. Boolean terms in path conditions are simplified in the same way. In this paper we limit ourselves to the case when path conditions involving new types can be simplified by means of TRS to relations containing only predefined types (and boolean True or False in the best case). So the solver is not supposed to find values of new types $t_1, t_2, \ldots$(except for trivial cases: any value and the value which has to be equal to some constant (literal) of new type).

Conditional rules in TRS (like in OBJ2 [24,27]) are also allowed, conditions should contain equalities (or inequalities) for predefined types, in particular, integers. If types and operators are generic, corresponding rules are also considered generic.

Let us consider an example: a new type queue of integers (it is used in a more general manner in Part 3). Let it have literal qnew and operators

```
qadd:integer, queue --> queue
qfirst: queue --> integer
qrest: queue --> queue.
```

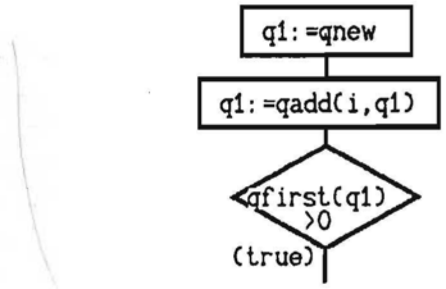
Then a standard form of TRS for this queue would be

```
qfirst (qadd (x, qnew)) --> x
qfirst (qadd (x1, qadd (x2, q))) --> qfirst (qadd (x2, q))
qrest (qnew) --> qnew
qrest (qadd (x,qnew)) --> qnew
qrest(qadd(x1,qadd(x2,q)))-->qadd(x1,qrest(qadd(x2,q)))
```

It can be simply deduced from the signatures that x, x1, x2

stand for integers, q for queue.

If we have a program fragment (with variable q1 declared as queue)



then we have at its symbolic execution:

q1=qnew (after statement1)

q1=qadd (i_1, qnew) (after statement2)

The true exit of the decision yields condition

qfirst (qadd(i_1,qnew))>0

which is reduced by simplifier (using the first rule for queues) to

i_1>0

(a condition completely manageable by the solver).

3. Symbolic Execution and Test Generation for Concurrent Programs

3.1 General Principles of Test Generation for Communicating Processes

In this part we consider symbolic execution and automatic test generation for real time programs in the specification language SDL. Our investigations are demonstrated for a subset of SDL including all essential concepts of the language used to describe parallel processes. We consider open systems having one or more channels from environment to system (and possibly some channels to environment). A system can contain one or more blocks, a block can contain one or more processes, procedures are also permitted. Dynamic creation of process instances is not included and all signals are assumed to be sent via channels and signalroutes. Each process is assumed to have only one instance, interprocess communication is solely by signals, viewing/revealing and export/import are not considered. We also

don't consider enabling conditions and continuous signals.

A test for an SDL system is a completely ordered sequence of input signals (including their parameter values) sent from environment to system through appropriate channels. If there are timers in the system, also the signal arrival times are fixed in the test (if there are no timers, only the order of signals is significant).

The main goal of our research is to construct complete test set (CTS) for an SDL system. A problem of completeness criterion arises just as for sequential programs. We accept the same criterion C1, nonetheless this time its use is not so obvious, besides, its definition requires some comments. By a branch in an SDL process we understand either an input branch starting from signal input statement or a conventional program branch starting from decision statement (or START statement). A branch ends at nextstate, decision or stop statement. Criterion C1 requires every feasible branch in every process to be executed at least once. Let us remark that sometimes more stringent criteria taking into account the concurrent nature of SDL processes are used, however, there is no such one widely adopted.

All our research in SDL area is demonstrated on a popular protocol example, namely, the sliding window protocol [25,26]. At first we describe the example itself. Then we define the symbolic execution of a path in SDL system (defining at first the path itself in some reasonable way). We explain how to construct and solve path inequalities like in Sections 2.4 - 2.6. A method for constructing (potentially infinite) execution tree similar to ACT used in [14] is given. A heuristic state-based approach to construct CTS using an initial segment of this tree is briefly described, CTS for the sliding window example can be built by means of this approach. To improve the performance of CTS construction algorithms we outline the main ideas of a more sophisticated approach which uses the symbolic execution of separate processes more deeply and allows to construct CTS for this example (and even more realistic protocols) while keeping the search in reasonable limits acceptable for practice.

3.2 Sliding Window Protocol Example

Sliding window protocol is a popular error recovery technique used in many real protocols at data link layer. At first we present its informal description taken from [25].

3.2.1 Overview

The sliding window protocol supports unidirectional message flow from transmitter to receiver with positive acknowledgement sent back on each transfer. Windows are used for flow control in both transmitter and receiver. The protocol operates over a medium which may lose, reorder or corrupt messages and acknowledgements. It is assumed that corruption of messages can be reliably detected by protocol using checksums sent with messages.

3.2.2 Sequence Numbering

The transmitter sends a sequence number with each message. A sequence number is unbounded and is incremented for each new message. The first message transmitted is given sequence number 1.

The receiver sends an Acknowledgement when it receives a message. The Acknowledgement carries a sequence number which refers to the last message successfully transferred to the receiving user. If an Acknowledgement has to be sent before a successful reception (e.g., the first message was corrupted), it is given sequence number 0.

3.2.3 Transmitter Behaviour

The transmitter maintains a window of sequence numbers as shown in Figure 3.1.

This gives the lowest sequence number for which an Acknowledgement is awaited, and the highest sequence number so far used. The window size is limited to the value tws.

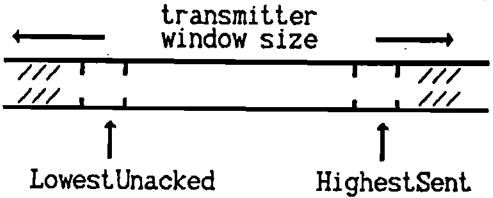


Figure 3.1. Transmitter Window Parameters

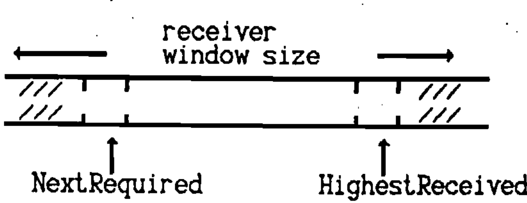


Figure 3.2. Receiver Window Parameters

The transmitter behaves initially as (a) below, and then loops doing (b), (c) and (d) where possible:

(a) LowestUnacked is set to 1 and HighestSent to 0

(b) If the current window size (HighestSent-LowestUnacked+1) is less than tws, then a message with the next sequence number (HighestSent+1) may be transmitted. In this case, HighestSent is incremented, and a timer for that message is started.

(c) If an Acknowledgement is received which is not corrupted and whose sequence number is not less than LowestUnacked, then all timers for messages up to and including that sequence number are cancelled. In this case, LowestUnacked is set to the sequence number following the acknowledged one.

(d) If a time-out occurs, then the timers for all messages transmitted after the timed-out one are cancelled. All these timed-out messages are retransmitted (in sequence, starting with the earliest) and have timers started for them.

3.2.4 Receiver Behaviour

The receiver maintains a window of sequence numbers as shown in Figure 3.2.

This gives the lowest sequence number which is awaited NextRequired and the highest sequence number which has been received. The window size is limited to the value rws.

The receiver behaves initially as (a) below, and then loops doing (b) and (c) where possible.

(a) NextRequired is initialized to 1

(b) If a message is received which is not corrupted, which has not already been received and which lies within the current receive window (NextRequired +rws-1), then all messages from NextRequired up to but not including the first unreceived message are delivered to the receiving user. (There may be no such messages if there is a gap due to misordering). In this case, NextRequired is set the sequence number of the next message to be delivered to the receiving User.

(c) If a message is received under any circumstances, an Acknowledgement giving the last delivered sequence number (NextRequired-1) is returned.

3.2.5 SDL Description of Protocol

SDL description of the protocol is also taken from [25]. Some obvious errors are corrected and medium description is slightly changed to adapt it for testing purposes.

The description consists of three blocks representing sender, receiver and medium. Both protocol user supplying data for sender and user consuming data from receiver are located in the environment. The sending user supplies data via channel ut by signals UDTreq, the receiving one gets data via channel ur by signals UDTind. The sender forms messages from each data unit (signal MDTreq) and passes them to medium via channel mt, acknowledgements (MAKind) are received from medium via the same channel.Conversely, the receiver gets messages from medium (MDTind) and puts acknowledgements (MAKreq) onto it via bidirectional channel mr.

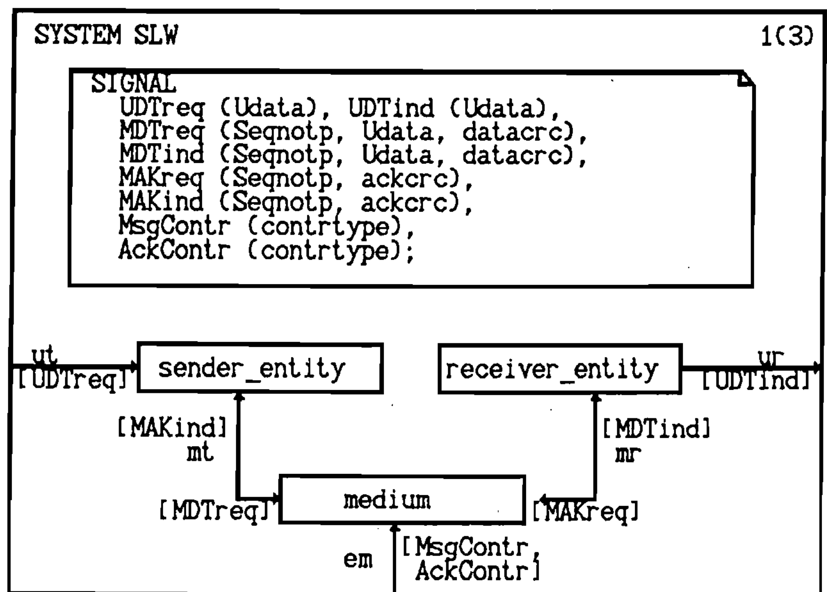
The Sender_entity block contains one process Transmit performing all sending actions. Each message (MDTreq) sent contains the generated sequence number, user data (of some unspecified type Udata) and cyclic range check computed by function dcheck. Data in transmitter window (i.e., sent but not acknowledged) are represented by queue mq, the current window limits are held in variables lu and hs. Time-out management is accomplished by setting indexed timer tim

with the corresponding seqno parameter for every message sent (and resetting it when acknowledgement arrives). The timer parameter also shows which timer instance has expired (and which messages are to be resent respectively). The time-out value is some constant delta. When the window contains maximum number of messages, the process enters the second state window_closed.

The receiver_entity block contains one process Receiver. The Next-Required sequence number is held in nr, message data received out of order (within window) are held in the array recbuf, the boolean array already_rec (of the same size) records which messages have arrived (but have not been delivered to the user yet).

The medium block contains processes MsgMan and AckMan managing the message and acknowledgement queues respectively. Message queue actions (normal transfer of message, loss of first message, reordering of messages in queue, corruption of the first message) are controlled by corresponding orders from system tester (signal MsgContr) sent from the environment. We note that in [25] the equivalent signals are generated randomly.

In the case of normal transfer the medium actually performs only signal renaming (from MDTreq to MDTind) while retaining the same parameters. Message corruption is performed by special function corrm. Acknowledgement queue manager performs the same way.

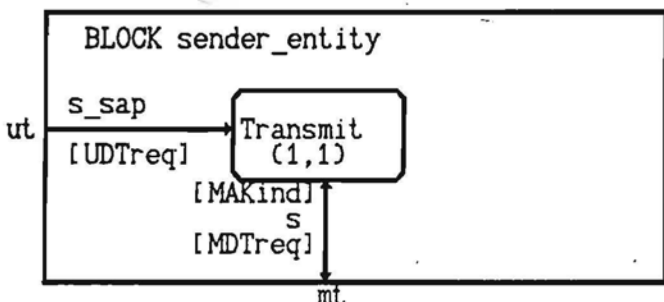


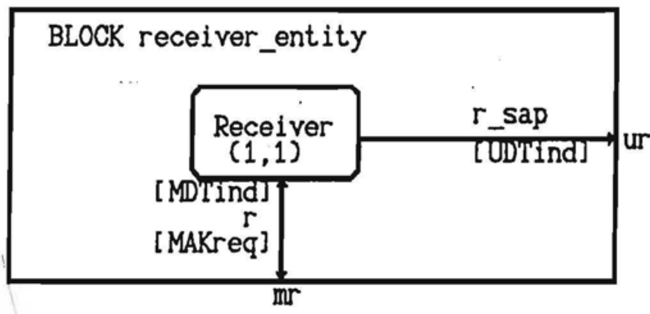
SYSTEM SLW 2(3)

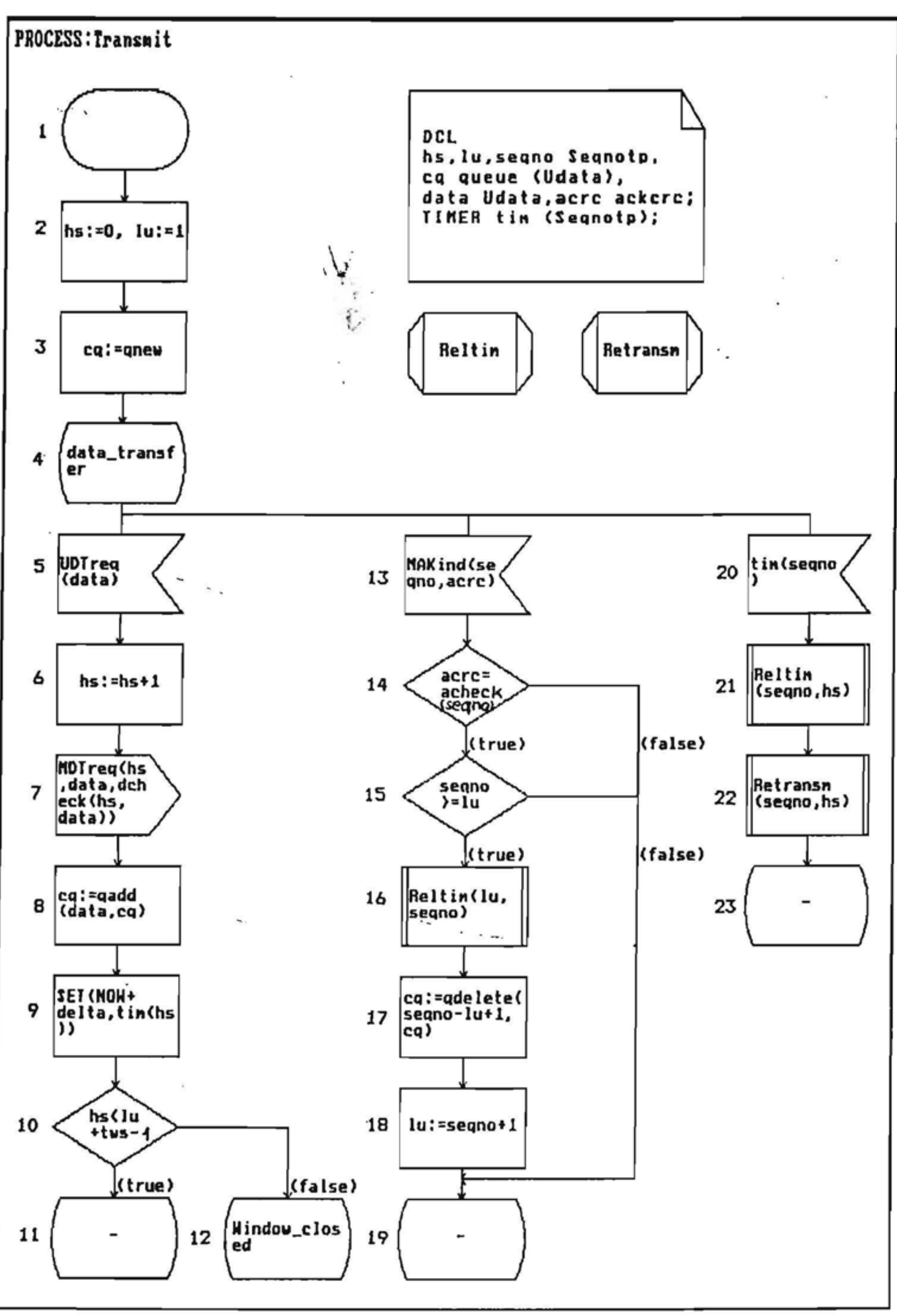
```
NEWTYPE Udata
ENDNEWTYPE Udata;

SYNTYPE Seqnotp=INTEGER
ENDSYNTYPE Seqnotp;

NEWTYPE datacrc
OPERATORS dcheck: Seqnotp, Udata ⟶ datacrc
/*builds crc field for a given pair of sequence
number and userdata in data message */
ENDNEWTYPE datacrc;

NEWTYPE ackcrc
 OPERATORS acheck: Seqnotp ⟶ ackcrc
/* builds crc field for a sequence number in ack-
nowledgement */
ENDNEWTYPE ackcrc;

NEWTYPE contrtype
  LITERALS norm, lose, reord, corr;
/* tester control options for medium action */
ENDNEWTYPE contrtype;

SYNONYM tws NATURAL = EXTERNAL;
SYNONYM rws NATURAL = EXTERNAL;
SYNONYM delta REAL  = EXTERNAL;
/* external parameters of the system */

 GENERATOR queue (TYPE item);
 LITERALS qnew;
 OPERATORS
        qadd: item, queue ⟶ queue;
       qfirst: queue ⟶ item;
       qrest: queue ⟶  queue;
       qdelete: integer,queue ⟶ queue;
       qreplace: item, queue ⟶ queue;
       qempty: queue ⟶ BOOLEAN;
AXIOMS
qfirst(qnew)== ERROR!;
qfirst(qadd(x,qnew))== x;
qfirst(qadd(x1,qadd(x2,q)))==qfirst(qadd(x2,q));
qrest(qnew)==qnew;
qrest(qadd(x,qnew))==qnew;
qrest(qadd(x1,qadd(x2,q)))==qadd(x1,qrest(qadd(x2,
                                                q)));
qempty(qnew);
NOT(qempty(qadd(x,q)));
qdelete(i,q)==IF i=0 THEN q
               ELSE qdelete (i-1,qrest(q))FI;
qreplace(x1,qadd(x2,qnew))==qadd(x1,qnew);
qreplace(x1,qadd(x2,qadd(x3,q)))==
        qadd(x2,qreplace(x1,qadd(x3,q)));
/*replaces the first element of queue by new value*/
 ENDGENERATOR queue;
```

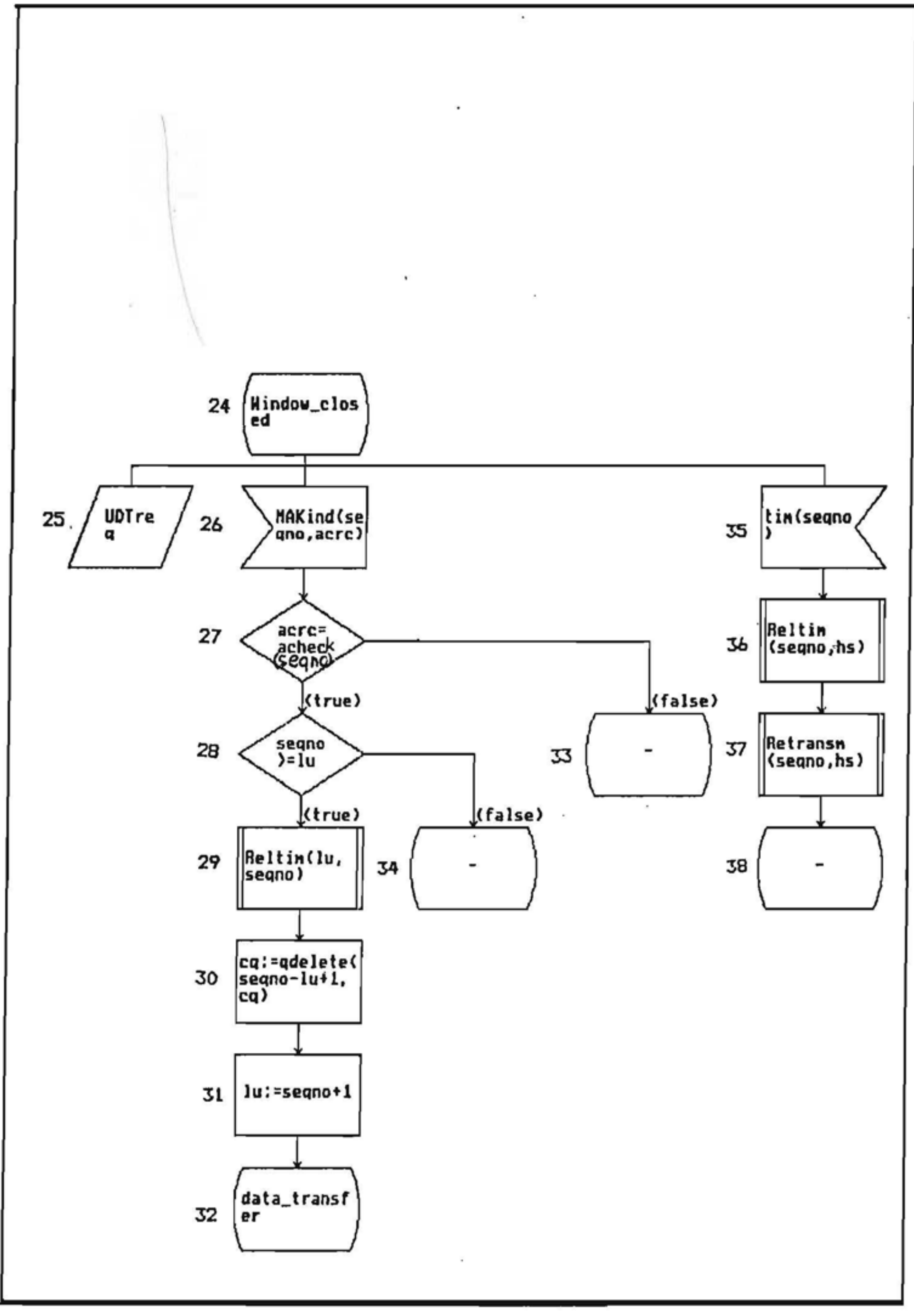
SYSTEM SLW 3(3)

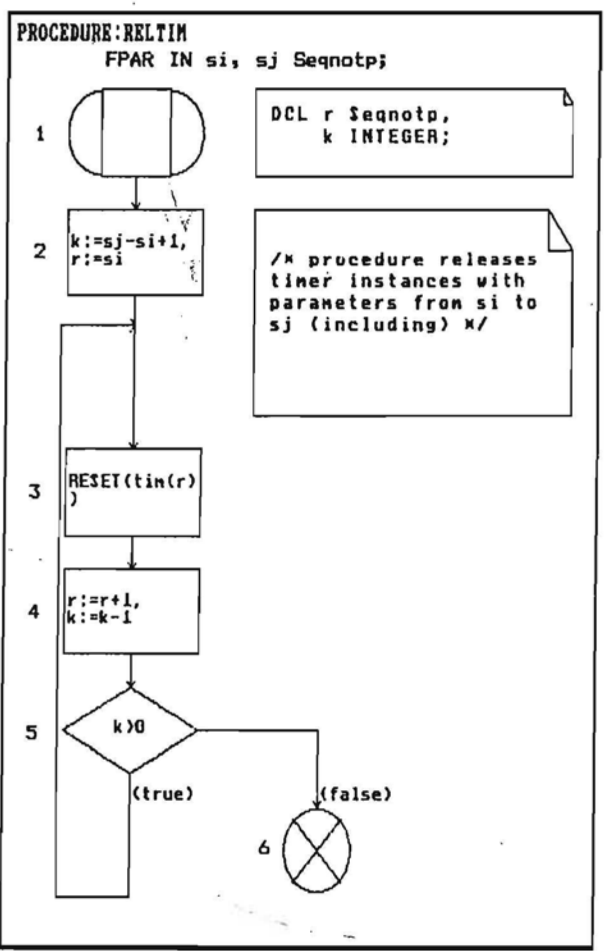
```
NEWTYPE message
      STRUCT
       seq Seqnotp;
       dat Udata;
       dc datacrc;
ADDING
OPERATORS
            corrm: message ⟶ message;
/* message corruption procedure */
  AXIOMS
   NOT (dcExtract!(corrm(m))=dcheck(seqExtract!
       (corrm(m)),datExtract!(corrm(m))));
/* every corruption is reliably detected by dcheck*/
   ENDNEWTYPE message;
NEWTYPE acknow
     STRUCT
       seq seqnotp;
       ac ackcrc;
ADDING
OPERATORS
     corra: acknow ⟶ acknow;
AXIOMS
 NOT(acExtract!(corra(a))=acheck(seqExtract!
                                  (corra(a))));
 ENDNEWTYPE acknow;
SYNTYPE rsn=INTEGER
         CONSTANTS 0:rws-1
         ENDSYNTYPE rsn;
```

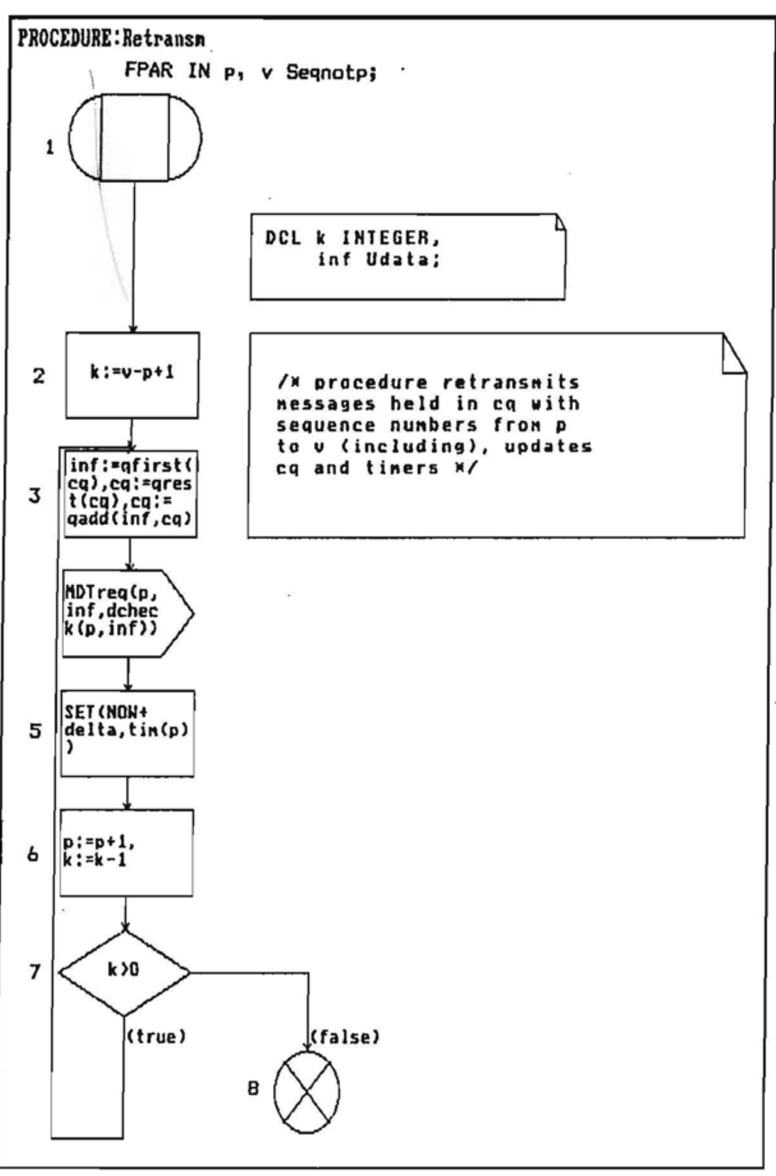


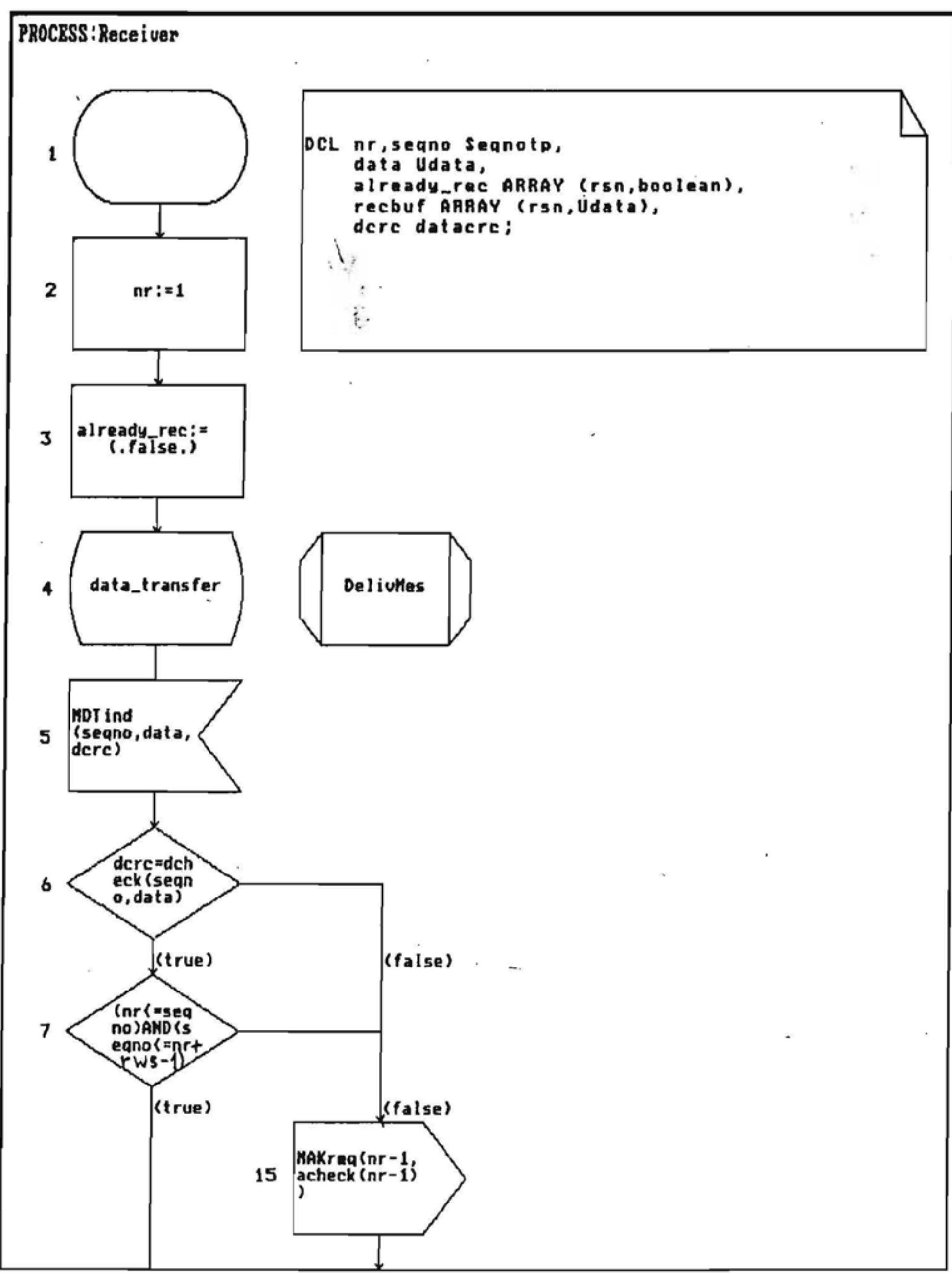

BLOCK receiver_entity
Receiver
(1,1)
r_sap
[UDTind]
ur
[MDTind]
r
[MAKreq]
mr

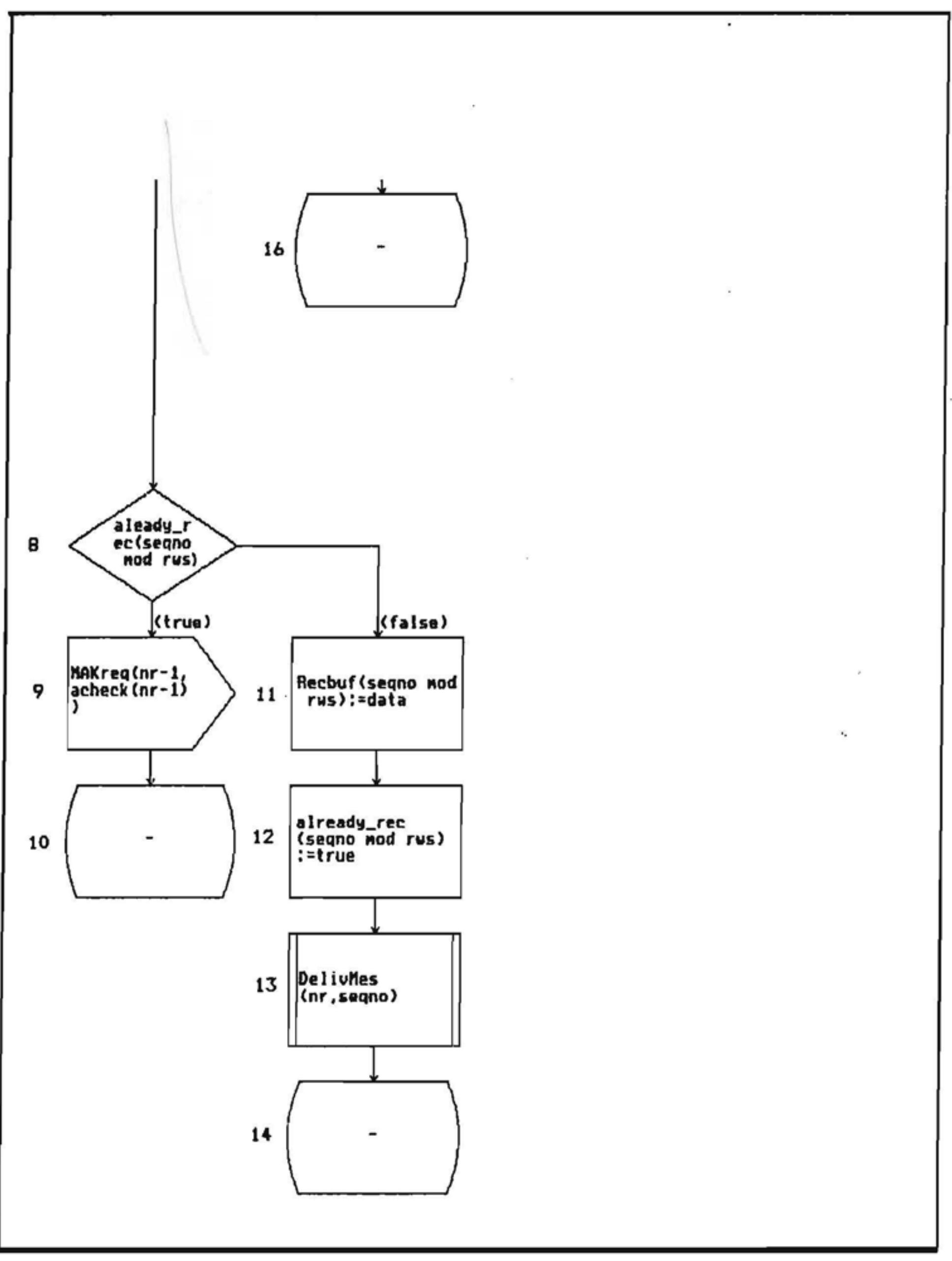

PROCESS:Transmit
DCL
hs,lu,seqno Seqnotp,
cq queue (Udata),
data Udata,acrc ackcrc;
TIMER tim (Seqnotp);
Reltim
Retransm
1
2
hs:=0, lu:=1
3
cq:=qnew
4
data_transf
er
5
UDTreq
(data)
6
hs:=hs+1
7
MDTreq(hs
,data,dch
eck(hs,
data))
8
cq:=qadd
(data,cq)
9
SET(NOW+
delta,tim(hs
))
10
hs<lu
+tws-1
(true)
(false)
11
-
12
Window_clos
ed
13
MAKind(se
qno,acrc)
14
acrc=
acheck
(seqno)
(true)
(false)
15
seqno
)=lu
(true)
(false)
16
Reltim(lu,
seqno)
17
cq:=qdelete(
seqno-lu+1,
cq)
18
lu:=seqno+1
19
-
20
tim(seqno
)
21
Reltim
(seqno,hs)
22
Retransm
(seqno,hs)
23
-


24
Window_clos
ed
25
UDTre
q
26
MAKind(se
qno,acrc)
35
tim(seqno
)
27
acrc=
acheck
(seqno)
36
Reltim
(seqno,hs)
(true)
(false)
28
seqno
)=lu
33
-
37
Retransm
(seqno,hs)
(true)
(false)
29
Reltim(lu,
seqno)
34
-
38
-
30
cq:=qdelete(
seqno-lu+1,
cq)
31
lu:=seqno+1
32
data_transf
er

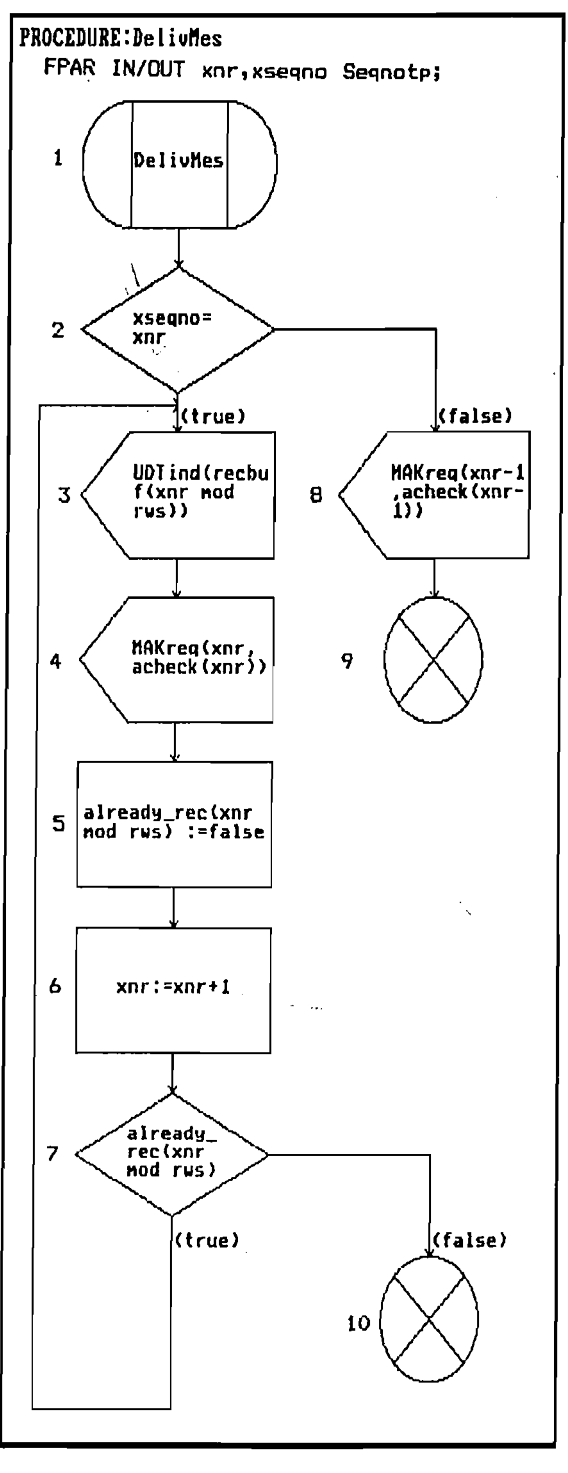

PROCEDURE:RELTIM
FPAR IN si, sj Seqnotp;
DCL r Seqnotp,
k INTEGER;
1
2
k:=sj-si+1,
r:=si
/* procedure releases timer instances with parameters from si to sj (including) */
3
RESET(tim(r))
4
r:=r+1,
k:=k-1
5
k>0
(true)
(false)
6

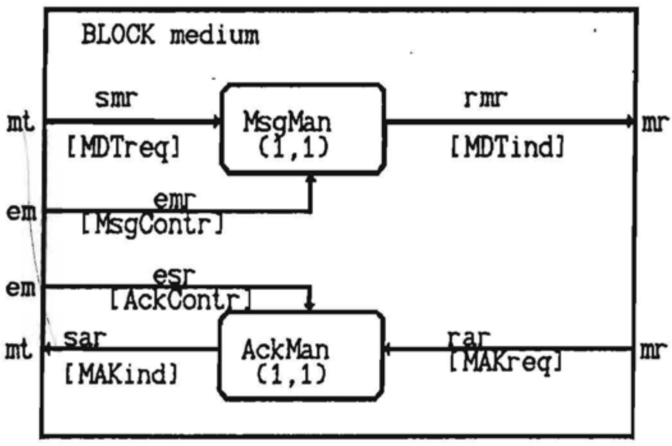

PROCEDURE:Retransm
FPAR IN p, v Seqnotp;
1
DCL k INTEGER,
inf Udata;
2
k:=v-p+1
/* procedure retransmits messages held in cq with sequence numbers from p to v (including), updates cq and timers */
3
inf:=qfirst(cq),cq:=qrest(cq),cq:=qadd(inf,cq)
MDTreq(p,inf,dcheck(p,inf))
5
SET(NOW+delta,tim(p))
6
p:=p+1,
k:=k-1
7
k>0
(true)
(false)
8

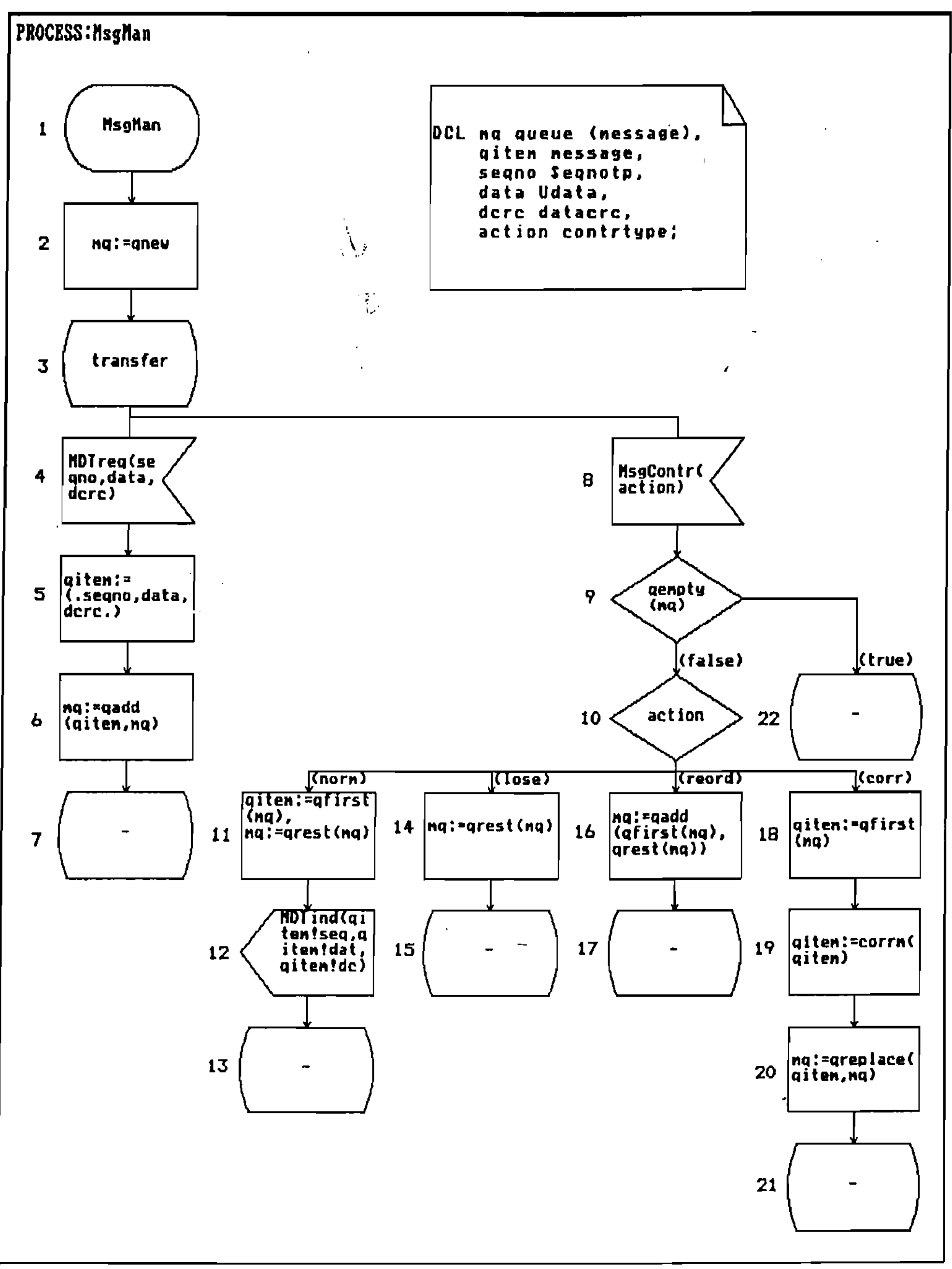

PROCESS:Receiver
1
2
nr:=1
DCL nr,seqno Seqnotp,
data Udata,
already_rec ARRAY (rsn,boolean),
recbuf ARRAY (rsn,Udata),
dcrc datacrc;
3
already_rec:=
(.false.)
4
data_transfer
DelivMes
5
MDTind
(seqno,data,
dcrc)
6
dcrc=dch
eck(seqn
o,data)
(true)
(false)
7
(nr(=seq
no)AND(s
eqno(=nr+
rws-1)
(true)
(false)
15
MAKreq(nr-1,
acheck(nr-1)
)

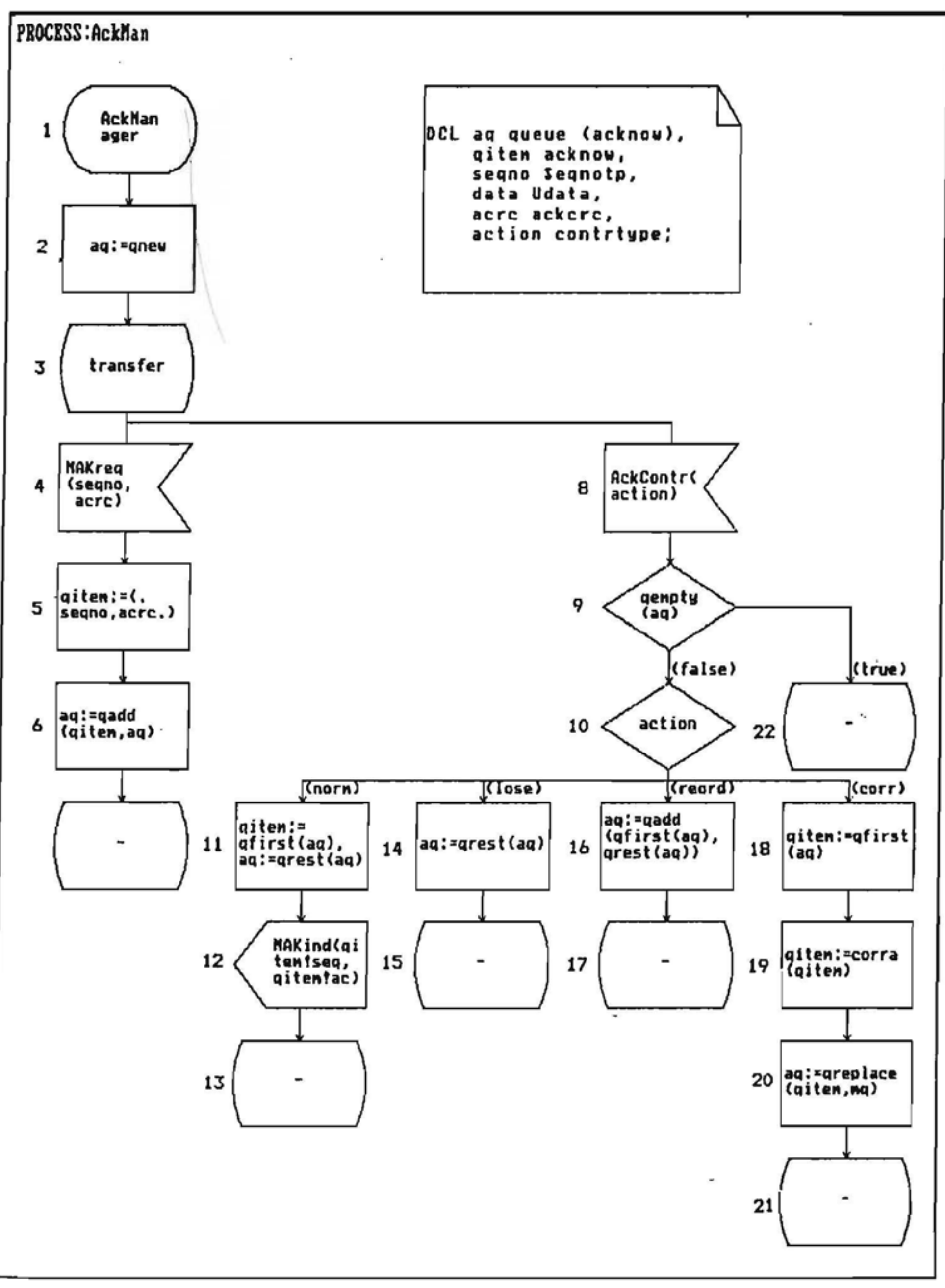


16
-
8
aleady_r
ec(seqno
mod rws)
(true)
(false)
9
MAKreq(nr-1,
acheck(nr-1)
)
11
Recbuf(seqno mod
rws):=data
10
-
12
already_rec
(seqno mod rws)
:=true
13
DelivMes
(nr,seqno)
14
-


PROCEDURE:DelivMes
FPAR IN/OUT xnr,xseqno Seqnotp;
1
DelivMes
2
xseqno=
xnr
(true)
(false)
3
UDTind(recbu
f(xnr mod
rws))
8
MAKreq(xnr-1
,acheck(xnr-
1))
4
MAKreq(xnr,
acheck(xnr))
9
5
already_rec(xnr
mod rws) :=false
6
xnr:=xnr+1
7
already_
rec(xnr
mod rws)
(true)
(false)
10


BLOCK medium
smr
mt
[MDTreq]
MsgMan
(1,1)
rmr
mr
[MDTind]
em
emr
[MsgContr]
em
esr
[AckContr]
mt
sar
[MAKind]
AckMan
(1,1)
rar
[MAKreq]
mr


PROCESS:MsgMan
MsgMan
mq:=qnew
transfer
DCL mq queue (message),
qitem message,
seqno Seqnotp,
data Udata,
dcrc datacrc,
action contrtype;
MDTreq(seqno,data,dcrc)
MsgContr(action)
qitem:=(.seqno,data,dcrc.)
qempty(mq)
(false)
(true)
mq:=qadd(qitem,mq)
action
(norm)
(lose)
(reord)
(corr)
qitem:=qfirst(mq),
mq:=qrest(mq)
mq:=qrest(mq)
mq:=qadd(qfirst(mq),qrest(mq))
qitem:=qfirst(mq)
MDTind(qitem!seq,qitem!dat,qitem!dc)
qitem:=corrm(qitem)
mq:=qreplace(qitem,mq)


PROCESS:AckMan
DCL aq queue (acknow),
qitem acknow,
seqno Seqnotp,
data Udata,
acrc ackcrc,
action contrtype;
1
AckMan ager
2
aq:=qnew
3
transfer
4
MAKreq (seqno, acrc)
5
qitem:=(. seqno,acrc.)
6
aq:=qadd (qitem,aq)
-
8
AckContr(action)
9
qempty (aq)
(false)
(true)
10
action
22
-
(norm)
(lose)
(reord)
(corr)
11
qitem:= qfirst(aq), aq:=qrest(aq)
14
aq:=qrest(aq)
16
aq:=qadd (qfirst(aq), qrest(aq))
18
qitem:=qfirst (aq)
12
MAKind(qitem!seq, qitem!ac)
15
-
17
-
19
qitem:=corra (qitem)
13
-
20
aq:=qreplace (qitem,mq)
21
-

Some notes have to be added with respect to testing of the system. At first external constants (synonyms) have to be fixed. Two of them:-tws and rws are very essential, since they control loops and array sizes. Some reasonable values (not too small to make some branches infeasible, not too large to make tests enormous) are to be selected. So we set both

tws = rws = 3.

The third constant delta is less essential, it can be fixed to value, e.g., 10, when it matters.

The other problem is abstract data types. As we have explained in 2.7, axioms should be replaced by some TRS to make the simplifier work with new data types. So we present the following TRS which is "compatible" with axioms, confluent and terminating (not for every set of axioms such TRS can be found). Rules are given for the generic type queue (with some nonstandard operations) and corruption /crc check of messages (acknowledgements)

```
qfirst(qadd(x,qnew)) -->x
qfirst(qadd(x1,qadd(x2,q)))-->qfirst(qadd(x2,q))
qrest(qnew)-->qnew
qrest(qadd(x,qnew))-->qnew
qrest(qadd(x1,qadd(x2,q)))-->qadd(x1,qrest(qadd(x2,q)))
qempty(qnew)-->true
qempty(qadd(x,q))-->false
qdelete(0,q)-->q
i>0 =>qdelete(i,q)-->qdelete(i-1,qrest(q))
qreplace(x1,qadd(x2,qnew))-->qadd(x1,qnew)
qreplace(x1,qadd(x2,qadd(x3,q)))-->
          qadd(x2,qreplace(x1,qadd(x3,q)))
eq(dcExtract!(corrm(m)),dcheck(seqExtract!
       (corrm(m)),datExtract!(corrm(m))))-->false
eq(acExtract!(corra(a)),acheck(seqExtract!
                            (corra(a))))-->false
        /* eq is the equality relation */
```

(Here and further we use standard SDL syntax for struct extract functions, namely, <field_name>Extract!, not the one used in 2.3.).

3.3 Semantic Constraints on SDL Subset

We assume in general that SDL system is executing according to semantics of SDL-88 [15]. However, some inessential limitations and

changes are introduced to make the description of process of test generation more understandable. These changes are inessential for the example considered and, as we hope, for protocol specification in general.

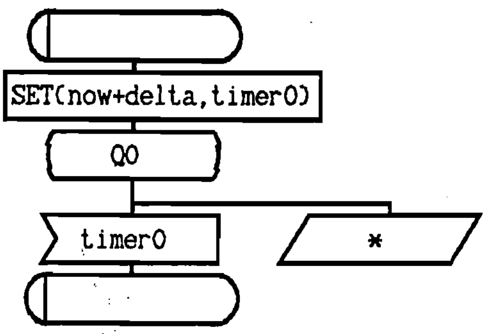
First, no two events in the whole system are assumed to be simultaneous, thus the events can be completely ordered in time. Actually, we make an even stronger assumption that only one transition from state to state occurs in the whole system at a given moment of time, the transition is always completed before another one takes place.

Second, all SDL actions including signal sending inside the system are assumed to be executing zero time. Thus, if a signal is sent from one process to other (including sending via channel), the receiving process is ready to operate just after the sending process has completed its transition, no time advancement occurs at that operation. Time is advanced only at reception of every signal from environment, and at active timer "firing".

Third, "internal" signals have priority before the signals from environment, i.e., whilst some process queue is nonempty (except the case when all existing signals are saved in the current state), no signal from environment is permitted.

The abovementioned semantic restrictions allow to assume that the whole system is executing under the control of some nondeterministic scheduler, which chooses at random an active process (i.e., a process with nonempty queue containing nonsavable signals) and activates it for one transition.If no process is active, the scheduler allows either an environment signal to arrive or an active timer to "fire" (if there is such). At the very beginning of execution the scheduler activates the initial transitions of all processes one after another.

The semantics considered is very appropriate for test generation and deterministic testing in general. To confirm practical reasonability of it we note that deterministic testing of a protocol specification makes similar assumptions as a rule [13,14]. If there are time-consuming operations in process diagrams (making zero time unrealistic), explicit timing should be introduced. We recommend delay (delta) statement for this purpose (it is in fact a macro call for the following macrodefinition:



which is completely within our SDL subset.)

Signal propagating delays along "real channels" should actually be described explicitly as testing environment controlled medium description processes in the system (e.g., MsgMan and AckMan processes in our example) in order to make delay dependencies actually testable.

3.4 Symbolic Execution of SDL Programs (many communicating processes)

Now, as we have discussed our semantic restrictions of SDL system behaviour, we can define the symbolic execution of a path in SDL system.

At first we have to explain what is a path in a concurrent system like SDL. We rely strongly on our semantic restrictions and the notion of the nondeterministic scheduler. Informally, a path is a particular execution trace of an SDL system. To be more formal, a path is a sequence of transition segments where each transition segment is a path from state to state (start to state, state to exit) (containing no state inside) in some process. If the path contains a procedure call, a path fragment inside the called procedure body has to follow immediately (separated into several transition segments if states are entered).If transition segments A1,A2,... referring to the same process P are singled out from path, then Ai must lead to the same state Si from which Ai+1 begins. Pseudo - transition segments (save for an environment signal, implicit transition, i.e., signal consumption in a state where it is not awaited) are also admitted in the path where they are possible according to SDL semantics. Some additional choices refining the

path will be described in the course of symbolic execution.

If the order of transition segments in the path were chosen at random, it might be highly probable that SDL semantics were violated, e.g., consumption of a signal would be required when no signal has been sent to the process. Therefore the notion of an admissible path is introduced. Informally an <u>admissible path</u> is one which complies with finite automata properties of SDL semantics and the scheduling principles described above, e.g., a signal can be consumed only if there is such in the corresponding queue, a timer can "fire" only after it has been set, etc. The admissibility of a path can be checked formally, but this check can be performed only along with the symbolic execution of the path. On the other hand, symbolic execution is defined only for admissible paths. We define a joint procedure for admissibility check and symbolic execution of a given path. The procedure is halted when the path is not admissible. Let us point out that admissibility does not imply feasibility, it is only a prerequisite for it.

Now let us describe the admissibility check and symbolic execution algorithm. The admissibility check is defined in the form of admissibility rules to be applied to the current symbolic state. Let an SDL system S containing processes P1,P2,...,Pn be given and α be a path in S. Two new "implicit" variables Q(P) and T(P) are introduced for every process P, and the symbolic values of these variables are maintained. Informally, Q(P) is the signal queue for process P, and T(P) is its active timer set. Symbolic values of Q(P) are finite sequences of symbolic values of signals denoted as <S1,...,Sk>, values of T(P) are sets of symbolic values of timers {T1,...,Te}.

Let us begin with the description of admissibility check and symbolic execution for SDL systems without timers (as we have mentioned before, the symbolic execution and test generation is simpler in that case). So the variable T(P) will not be used for a while.

Symbolic execution is performed for every SDL statement, while admissibility check is performed only at the beginning of transition segment, i.e., when interpreting its state and input (or save) statements. In the beginning of the algorithm Q(P) are empty for all P, i.e., they contain empty signal sequence < >. Let us assume that α contains transition segments $A_1^{i_1}, A_2^{i_2}, A_3^{i_3}, \ldots,$ from processes $Pi_1, Pi_2, Pi_3, \ldots,$ respectively. For the moment we are interested in

and symbolic values of parameters are assigned to corresponding variables, e.g., x1 assumes symbolic value t_1^s. In the case of "internal input" the symbolic value of signal is obtained from the symbolic value of queue, i.e., the signal instance to be consumed is found in the corresponding signal sequence and after assignment this instance is deleted from the queue. In the case of "external input", as it was described earlier, new symbolic signal value $S(S_i^1,S_i^2)$ is generated (i is the number of instance of signal S sent from the environment). In the case of saving an ENV signal its symbolic value is added to the end of queue; implicit transition means the discarding of symbolic signal value.

Let us remark that generation of symbolic values for queues and signal consumption could be formalized by some TRS (using conditional rules), but we think this would add no clarity to our explanation.

Before proceeding to an example we note that a "very short" form of symbolic language is used to improve readability (type postfixes omitted at all, trivial path conditions from range checks not included).

Let us show an example of symbolic execution of a path in our system SLW. Although actually it is a system with timers, we ignore them for a moment (omitting the setting statement 9). To indicate a path we use numeric labels of statements preceded by the first letter of process name (T for Transmit, R for Receiver, M for MsgMan,A for AckMan). Exits of decision statements are not indicated explicitly (they can be deduced from the next statement label). So, let us consider an initial path
T1,T2,T3,T4,R1,R2,R3,R4,M1,M2,M3,A1,A2,A3,T4,T5,T6,T7,T8,
T10,T11,M3,M4,M5,M6,M7.

The presence of start transitions (T1,T2,T3,T4,...) for all processes in the beginning of the path was required by admissibility rules (certainly, the order is inessential). As initial transitions contain no statements of "genuine SDL", the symbolic execution proceeds the same way as in Part 2.

So we present the symbolic state after the path T1,T2,T3,T4,R1,R2,R3,R4,M1,M2,M3,A1,A2,A3 at once. All symbolic values are shown to be simplified as far as possible by the simplifier described in 2.4

Transmit	Receiver	MsgMan	AckMan
hs=0	nr=1	mq=qnew	aq=qnew
lu=1	already_rec= (.false,false,false.)	qitem=undef	qitem=undef
cq=qnew	Q(Receiver)=<>	action=undef	action=undef
seqno=undef	recbuf=undef	Q(MsgMan)=<>	Q(AckMan)=<>
data=undef			
acrc=undef			
Q(Transmit)=<>			

Path condition: true

Some variables with undef values are not shown.

Statements T4,T5,...T11 form the first nontrivial transition segment (in the process Transmit). It conforms to admissibility rules since all queues are empty and "external input" occurs (namely, ENV signal UDTreq enters). After statement T5 the symbolic value of variable data is updated

$$data = UDTreq_1^1$$

(a new symbolic initial value has been generated, involving the first instance of UDTreq).

Statement T6 also updates one value

hs=1.

Statement T7 updates the queue value of process MsgMan, since channels and routes direct the signal MDTreq to this process

Q(MsgMan)=<MDTreq(1,$UDTreq_1^1$,dcheck(1,$UDTreq_1^1$))>.

Statement T8 adds the following

C_q=Qadd($UDTreq_1^1$,qnew).

Decision statement T10 adds no path condition since its value 1<1+3-1 is reduced to true by simplifier, the "true" exit implicitly assumed in the path is valid. Statement T11 closes the transition by returning to state Data-transfer.

Now let us consider the second transition segment M3,M4,M5,M6,M7. As the queue Q(MsgMan) is nonempty (the other queues being empty), this is the only transition segment permitted by admissibility rules in this situation. The statement M4 updates the values of variables mentioned in this input statement

seqno=1

data=$UDTreq_1^1$

dcrc=dcheck(1,$UDTreq_1^1$).

Statement M5 forms new struct value

qitem=(.1,$UDTreq_1^1$,dcheck(1,$UDTreq_1^1$)).

After M6 we have

$mq=(qadd((.1,UDTreq^1_1,dcheck(1,UDTreq^1_1).),qnew)$.

The final symbolic state after the path is:

Transmit

hs=1
lu=1
$cq=qadd(UDTreq^1_1,qnew)$
seqno=undef
$data=UDTreq^1_1$
acrc=undef
Q(Transmit)=<>

MsgMan

$mq=qadd((.1,UDTreq^1_1,dcheck(1,UDTreq^1_1).),qnew)$
$qitem=(.1,UDTreq^1_1,dcheck(1,UDTreq^1_1).)$
action=undef
Q(MsgMan)=<>

Receiver

nr=1
already_rec=(.false,false,false.)
recbuf=undef
Q(Receiver)=< >

AckMan

aq=qnew
qitem=undef
action=undef
Q(AckMan)=<>

The path condition remains true.

The path occurs to be both admissible and feasible.

Now let us consider the general case when timers are used. In this case the active timer set T(P) is maintained during the symbolic execution for every process P and admissibility rules for timers rely on this set. The initial value of T(P) is empty set {}. Timer instances (or, more precisely, symbolic values of timers) are added to the set by SET statements. The symbolic value of the timer consists of its name followed by the symbolic value of time moment to which the timer is set (and symbolic values of parameters, if there are such), for example, $tcon(t^s)$, $tim(t^s_1,1)$. The set T(P) contains at most one instance of each timer in the process P (in the case of timers with parameters, one instance for each distinct value of parameters).

Admissibility rule for timers says that "timer transition" (i.e., transition starting with timer input) is permitted only in "external input" situation if the corresponding timer instance is in T(P) for process P under consideration. To define the symbolic execution of time involving statements, a new, real valued variable NOW is introduced (one for the whole system). The initial value of NOW is 0, and it contains the symbolic value of system time at every moment (as demanded by SDL semantics).

Basic "reference points" for time counting are times of arrival of ENV signals. Every instance of signal S sent from the environment has associated its symbolic arrival time value S_i^T (i is the instance number just as for initial values of parameters). Values of the form S_i^T (for all ENV signals) play the role of initial symbolic values for time counting. When the input of ENV signal S is executed, the symbolic value of NOW is set to S_i^T. The old symbolic value of NOW (i.e., before the new assignment, let us denote this value by NOWold) is used to add a new inequality

$$\mathrm{NOWold} < S_i^T$$

to path condition. The inequality expresses the fact that according to our modifications of SDL semantics a new ENV signal cannot be simultaneous with some previous event in the system. The saving of ENV signal advances NOW in the same way.

For example, if we consider the previous example as a system with timers (in fact, it is such), then after statement T5 the value of NOW is

$$\mathrm{NOW}=\mathrm{UDTreq}_1^T$$

and the inequality $0<\mathrm{UDTreq}_1^T$ is added to path condition (the previous value of NOW was the initial value 0).

Next we consider the symbolic execution of SET statement. This statement has the form SET(t,tim), where t is an expression of type real (as a rule, in the form NOW+t_1, t_1 can also be an expression of type real but often is a constant) and tim is a timer name. At first the symbolic value of t (denoted by t^s) is obtained. Then the symbolic value of timer tim(t^s) is added to T(P), where P is the current process.

If there already is an instance of tim in T(P), the old instance is removed. For a timer with parameters the action is similar. Let us consider SET statement SET(t, tim1(p_1)). At first we assume that expression p_1 can be reduced to some constant C_1 (of the corresponding type) when computing its symbolic value p_1^s. Then

$tim1(C_1)$ acts in fact as an independent timer. We also assume instances of tim1 in T(P) having the same property that their parameters are reduced to constants. So symbolic value $tim1(t^s,C_1)$ is added to T(P), and, if there is an instance $tim1(t'^s, C_1)$ with the same constant parameter in T(P), the previous one is deleted.Let us return to our example and restore statement T9, omitted at first.

Let us remind that before T9 the value of NOW is $UDTreq_1^T$ and HS=1. Then after statement T9 we have a timer value

$$tim(UDTreq_1^T+delta,1)$$

and T(Transmit) assumes value

$$\{tim(UDTreq_1^T+delta,1)\}.$$

Now we have to consider the most general case when either the parameter value p_1^s for the timer tim1 to be set cannot be reduced to constant by simplifier or T(P) already contains an instance of tim1 with non-constant parameter. Let us assume the instances of tim1 in T(P) to be

$$tim1(t_1,q_1),\ tim1(t_2,q_2),\dots,tim1(t_k,q_k)$$

(q_i are symbolic values of the parameter).

In this moment path refinement is done. The following cases are possible here - either the new symbolic value of the parameter p_1^s coincides with one of the existing values, say, q_j, or p_1^s is a new value. Path refinement means an a priori choice of one of the possibilities (it is reasonable to call this choice a path refinement because admissibility of timer transitions later on the path depends on the choice). If the first case is chosen, $tim1(t^s,p_1^s)$ is added to T(P), $tim1(t_j,q_j)$ is removed, and besides that equality

$$p_1^s = q_j$$

is added to path condition. If the second case is chosen, $tim1(t^s,p_1^s)$ is added to T(P) and inequalities

$$p_1^s \neq q_1,\dots,p_1^s \neq q_k$$

are added to path condition.

The symbolic execution of RESET statement is similar. The corresponding instance of the timer is simply removed from T(P) when the timer has no parameters or all parameters of timer instances (i.e., their symbolic values) can be reduced to a constant. In general case for timer resetting with parameters a similar path refinement is made and corresponding equalities (inequalities) are added to path condition.

The using of timers involves additional timing constraints in

path condition. So, when active timer set T(P) is nonempty for at least one of the processes P, additional inequalities are to be added to path condition at ENV signal input. The new symbolic value of NOW (namely, S_i^T if signal S is consumed the i-th time) has to be less than the value of time held in any instance of active timer, so inequalities

$$S_i^T < t_j$$

are added to path condition for symbolic value of time t_j held in any active timer instance in any of $T(P_i)$.

Let us consider an example. We extend the path considered in the previous example (with statement T9 reinserted) the following way:

T1,T2,T3,T4,R1,R2,R3,R4,M1,M2,M3,A1,A2,A3,T4,T5,T6,T7,T8,
T9,T10,T11,M3,M4,M5,M6,M7,T4,T5,T6,T7,T8,T9,T10,T11.

We describe completely the symbolic execution of the second occurrence of T5. We have before it

hs=1

$T(Transmit)=\{tim(UDTreq_1^T+delta,1)\}$

$NOW=UDTreq_1^T$

The execution of T5 gives

$NOW=UDTreq_2^T$

and two new inequalities in the path condition

$UDTreq_1^T<UDTreq_2^T$

$UDTreq_2^T<UDTreq_1^T+delta$

After the second occurrence of T9 we have

$T(Transmit)=\{tim(UDTreq_1^T+delta,1),tim(UDTreq_2^T+delta,2)\}$

The last item to be described is the symbolic execution of timer "firing". In process P the symbolic execution of timer input, i.e., statement

tim(x) ,

invokes the following actions. At first an instance of timer tim is selected in T(P), let it be $tim(t^s,p^s)$ (for timers without parameters, it is the only instance of the timer, its existence is guaranteed by admissibility rules). The act of timer instance selection again is a path refinement. Then the symbolic value of NOW is set to t^s, the selected timer instance is excluded from T(P) and x assumes the value p^s. New inequalities expressing the fact that

time is nondecreasing and the timer with the least time value should "fire" the first are added to path condition. So inequalities

$$NOWold \le t^s$$

and

$$t^s \le t_j$$

for all symbolic values of time t_j held in any (remaining) active timer instance in any of $T(P_1)$. Inequalities are nonstrong this time because two timers can be set on the same time moment.

Now we give an example of timer "firing" which is the continuation of the previous example with the following two transitions added:

M3,M4,M5,M6,M7,T4,T20,T21,RL1,RL2,RL3,RL4,RL5,
RL3,RL4,RL5,RL6,T22,...

(RL stands for RelTim).

The "internal input" transition M3...M7 is implied by admissibility rules (the queue Q(MsgMan) is nonempty). This occurrence of the transition is similar to the first one and affects only variables in process MsgMan, so it is not described. We start the description with T20. The previous example shows that before it there holds

$NOW=UDTreq_2^T$
$T(Transmit)=\{tim(UDTreq_1^T+delta,1),tim(UDTreq_2^T+delta,2)\}$
$hs=2$,

all queues are empty. So T20 is admissible, we can select the timer instance to "fire". We choose the first one. After the execution of T20 we have

$NOW=UDTreq_1^T+delta$
$T(Transmit)=\{tim(UDTreq_2^T+delta,2)\}$
$seqno=1$

The following inequalities are added to the path condition

$$UDTreq_2^T \le UDTreq_1^T+delta$$
$$UDTreq_1^T+delta \le UDTreq_2^T+delta$$

Just the last inequality shows that our choice of timer instances is the only possible one to obtain a feasible path (and corresponds to reasonable behaviour of timers). Had we selected the second instance, we have had contradicting inequalities in path condition

$$UDTreq_1^T < UDTreq_2^T \quad \text{and}$$
$$UDTreq_2^T+delta \le UDTreq_1^T+delta,$$

the fact obviously noticed by our inequality solver. The next

statement T22 calls the procedure RelTim, so after statements RL1,RL2 we have

k=2

r=1

si=1

sj=2,

After RL3

T(Transmit)={tim($UDTreq_2^T$+delta,2)}

(no instance to reset actually). The path chosen in RelTim is the only feasible one in the given context, after second occurrence of RL3

T(Transmit)={ }

(because r=2 this time). So we can continue the execution, the path occurs to be feasible.

The defined timing inequalities have the property that path condition has a solution with respect to arrival times of ENV signals (i.e., the variables in the form Si_j^T) iff all events along the path can be allocated in time so that they comply with the details of SDL semantics laid out in Section 3.3. We could formulate this result as a theorem, had our description of symbolic execution been more formal.

We conclude this section by one more example, namely, we show the symbolic execution of another extension of the path considered in the first example. So we consider the path

T1,T2,T3,T4,R1,R2,R3,R4,M1,M2,M3,A1,A2,A3,T4,T5,
T6,T7,T8,T9,T10,T11,M3,M4,M5,M6,M7,M3,M8,M9,M10,
M11,M12,M13,R4,R5,R6,R7,R8,R11,R12,R13,D1,D2,D3,
D4,D5,D6,D7,D10,R14,A3,A4,A5,A6,A7,A3,A8,A9,A10,
A11,A12,A13,T4,T13,T14,T15,T16,RL1,RL2,RL3,RL4,RL5,
RL6,T17,T18,T19.

(Prefix D stands for procedure DelivMes, RL for RelTim).

This path corresponds to complete successful sending of one message from transmitter to receiver and successful acknowledgment sending vice versa. Active use of TRS to simplify symbolic values is demonstrated on the path. From the first example we know the symbolic state after M7:

<u>Transmit</u>

hs=1 T(Transmit)={tim($UDTreq_1^T$+delta,1)}

lu=1 Q(Transmit)=< >

cq=qadd($UDTreq_1^1$,qnew)

data=$UDTreq_1^1$
acrc=undef
seqno=undef

MsgMan

mq=qadd((.1,$UDTreq_1^1$,dcheck(1,$UDTreq_1^1$).),qnew)
qitem=(.1,$UDTreq_1^1$,dcheck(1,$UDTreq_1^1$).)
action=undef
T(MsgMan)={ }   Q(MsgMan=< >

Receiver

nr=1
already_rec=(.false,false,false.)
recbuf=undef
T(Receiver)={ }   Q(Receiver)=< >

AckMan

aq=qnew   T(AckMan)={ }
qitem=undef   Q(AckMan)=< >
action=undef

NOW=$UDTreq_1^T$

Path condition

$0<UDTreq_1^T$

External input in M3, M8 is obviously admissible.

We have after it

action = $MsgContr_1^1$

NOW = $MsgContr_1^T$,

path condition is augmented by

$UDTreq_1^T < MsgContr_1^T$

$MsgContr_1^T < UDTreq_1^T + delta$

From M9 with exit "false" we have condition

not(qempty(qadd((.1,$UDTreq_1^1$,dcheck(1,$UDTreq_1^1$).),qnew)),which is obviously reduced by a single TRS rule application to not(false)=true. So the feasibility of the selected path is not violated, no path condition is added.

Statement M10 (with exit norm implied) gives path condition

$MsgContr_1^1$ =Norm

After M11 we have

qitem= qfirst(qadd((.1,$UDTreq_1^1$,dcheck(1,$UDTreq_1^1$).),qnew))

evidently reduced by TRS to

qitem=(.1,$UDTreq_1^1$,dcheck(1,$UDTreq_1^1$).)

(namely the reduced value is fixed in symbolic value system), similarly the new value of mq is reduced to

mq=qnew.

Statement M12 augments the queue of Receiver

Q(Receiver)=<MDTind(1,$UDTreq_1^1$,dcheck(1,$UDTreq_1^1$))>

Further a nonempty queue for Receiver makes R4, R5 be the sole admissible continuation. After R5 we have in Receiver

seqno =1
data =$UDTreq_1^1$
dcrc =dcheck(1,$UDTreq_1^1$).

The exit true in statement R6 is, in fact, implied (dcheck(1,$UDTreq_1^1$)=dcheck(1,$UDTreq_1^1$) is reduced to true by the simplifier).

The chosen exit in the next two statements is also implied, for both

(1<= 1) AND (1<=1+3-1)

and

not(extract((.false,false,false.),1 mod 3))

reduces to true.

Statements R11 and R12 make

recbuf=modify(undef,1,$UDTreq_1^1$)
already_rec=modify((.false,false,false.),1,true)).

In the procedure DelivMes we have after D1

xnr =1
xseqno=1,

which implies the chosen exit of D2 (parameters are in/out, so the changed values are returned to nr, seqno). D3 sends signal

UDTind ($UDTreq_1^1$) to environment.

We can continue the symbolic execution of the path in the same way. All remaining decisions in the path uniquely reduce to true (except one in process AckMan which gives path condition

$AckContr_1^1$ =Norm). Admissibility rules uniquely determine the chosen internal transitions.

So we end the path with the following values (only the essential ones are given)

Transmit	MsgMan	Receiver	AckMan
hs=1	mq=qnew	nr=2	aq=qnew
lu=2		already_rec=	
cq=qnew		=(.false,false,false.)	

All sets T and queues Q are empty, NOW=$AckContr_1^T$,
final path condition is

0<$UDTreq_1^T$

$UDTreq_1^T < MsgContr_1^T$

$MsgContr_1^T < UDTreq_1^T + delta$

$MsgContr_1^1 = norm$

$MsgContr_1^T < AckContr_1^T$

$AckContr_1^T < UDTreq_1^T + delta$

$AckContr_1^1 = norm.$

The path is obviously feasible. The path condition requires only the arrival times of three ENV signals to be ordered properly, taking into account also the time-out interval delta. So the following ENV signal sequence UDTreq(data1),MsgContr(norm),AckContr(norm) with arrival times 1,2,3 (if delta is assumed to be, e.g., 10) is a test executing the chosen path (value of data1 is inessential).

So it is easy to ascertain that for every feasible path in the SLW example trivially solvable path conditions can be obtained. Due to this the corresponding ENV signal sequence (with their arrival times fixed) can be generated which actually forces the execution of the path.

3.5 Path Selection for Test Generation - Simple Approach

As we have seen in the previous section, it is logically easy (though a little bit lengthy) to find a test (ENV signal sequence) forcing the execution of a given feasible path. In order to obtain CTS for the system SLW it would be necessary to fix some path selection strategy. However, the paths considered in the previous section show a special feature of the system SLW (and this feature is common to many protocol and similar programs). Namely, the choice of feasible path continuations in decision statements is uniquely determined (i.e., a sole exit is feasible). The only exceptions are decisions relying upon ENV signal parameters. So the only free choice is the choice of ENV signal to be received (including parameters for signals to MsgMan and AckMan). The analysis of timing inequalities show that actually two things are significant - the order of arrival of ENV signals and whether the current ENV signal arrives before the time-out period has expired (for the timer set earliest). So the following choices are available at every "external input" point (in parenthesis the shorthand notation for the choice

is presented):

input of UDTreq (U),

input of MsgContr with parameter values norm (MN),
lose (ML), reorder (MR), corrupt (MC),

input of AckContr with parameter values norm (AN),
lose (AL), reorder (AR), corrupt (AC),

no ENV signal until the timer tim fires (T).

If we fix the ENV input string, the internal behavior of the system (and consequently, the path traversed in process bodies) is uniquely determined. As we have seen in the previous section, admissibility rules sometimes exclude T choice. The possible choices can be summarized in the following potentially infinite tree (if complete symbolic state remains unchanged after the choice, we cut off the tree after the branch). We call this tree an external signal tree (EST) (fig.3.3).

Any feasible branch in process bodies is executed somewhere in the tree. However, there is no good means to find out where exactly the point is in the tree. For example, to execute the branch R8, R9, R10, a path of length 7 in the tree is necessary (this branch seems to be the most "hidden").

As we see, the branching coefficient for the tree is 10 (excluding few first vertices), so direct exhaustive search of nearly 10^6 vertices would not be very efficient. State based theoretical methods from [17] are not directly applicable to the example (because of potentially unlimited queues), thus some heuristic methods are necessary to limit the search.

We outline briefly one such heuristic idea which uses the state notion as in [17], however, in a more heuristic sense. We recall the notions of essential variable and essentially located statement (ELS) introduced in [10,17], Section 3.

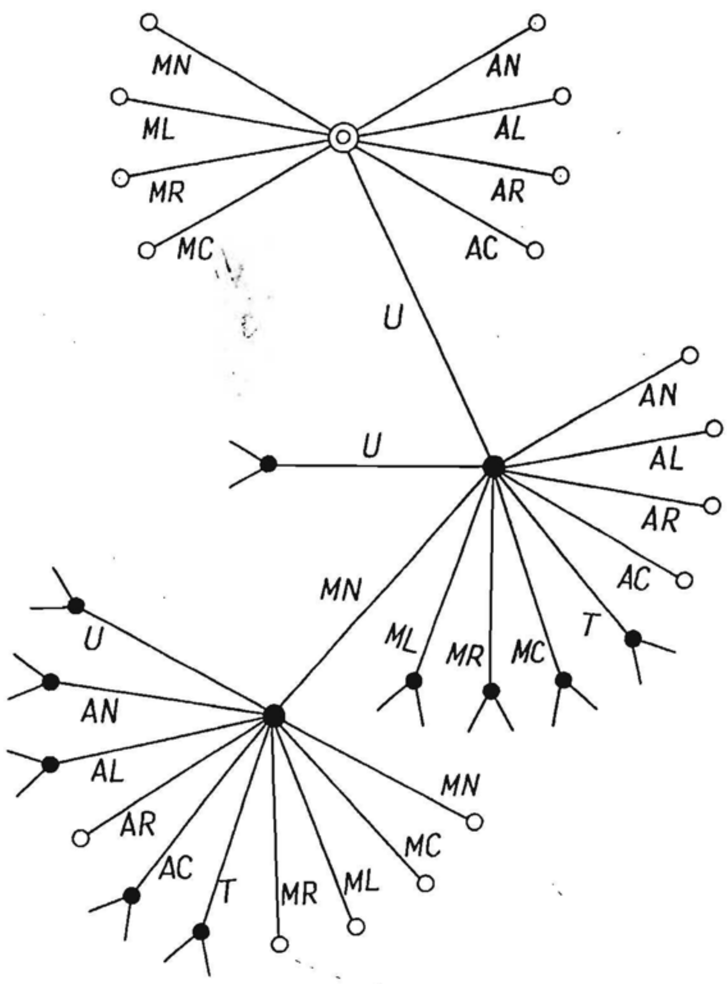


Figure 3.3. External Signal Tree

A comparatively simple analysis shows that there can be no unbounded loops within transition segments in our example. Moreover, only a bounded sequence of "internal" transition segments can follow an ENV signal input or timer firing. So we can choose the inputs of ENV and timer signals as ELS. Thus, essential variables are associated only with transitions corresponding to ENV and timer signals.

So, in a more pragmatic approach , we can say that a variable in a process is essential if it is used in decision statement and is not reassigned from ENV input to its usage in a decision (actually, for SLW example this requirement is equivalent to the formal one used in [17]. The variables affecting path admissibility are also considered essential. As we consider only external inputs as ELS, signal queues are not essential (they are always empty at these inputs), however timer sets are essential. So the following variables occur to be essential in our example: hs, lu, nr, already_rec, mq, aq, T(Transmit). Some additional arguments show that for elements of queues (mq and aq) only the sequence numbers and corrupted/not corrupted property is essential, so these elements can be reduced to pairs (s, 'n'|'c') in our state concept. For timer sets only the sequence numbers held as parameters are essential (the ordering of time moments is implied). So a reduced heuristic state containing only reduced values of essential variables is attached to each node of EST. And, as usually, the tree is cut off at state repetition, however, the finiteness of the set of states is not achieved this way. Some more stringent heuristic cut-off rules can be given, specific to this example, guaranteeing the finiteness of state set (e.g., replacing the counter values hs, lu, nr by some differences in state comparisons and estimating maximum lengths of mq, aq). A simpler heuristic approach is based on fact that all branches actually are reachable for hs, lu, nr and queue lengths not exceeding 4. So the estimated number of EST nodes to be searched for CTS building is approximately 1000 (cutting off nodes with variable values or queue lengths exceeding 4 and stopping the search when all branches have been reached). The introduced state concept actually also supports symbolic execution of each path, so the outlined approach can be used in tools generating CTS (a tool generating CTS for this example could be implemented on IBM PC). Essential variable selection and cut-off rules for states would be user supplied for

such tool. The heuristic state approach is practically acceptable for test generation for medium size protocols. A more efficient idea applicable to large systems is outlined in the next section.

To conclude the theme on signal tree we have to mention that EST is similar to asynchronous communication tree (ACT) used in [12,14]. ACT contains also signals from system to environment but our approach allows to find them as well (see signal UDTind in symbolic execution example). So the symbolic execution method allows to construct ACT also for protocols whose functioning depends essentially upon some data processing.

3.6 A More Intelligent Approach to Test Generation

As we have seen in the previous section, CTS can be generated on the basis of symbolic execution approach. However, though there are 34 branches (all feasible) in our example system, the state oriented approach requires considerable search (of ≈1000 states). When a human is asked to generate a test executing some branch he performs, as a rule, some backward search from the specified branch trying to find out gradually some meaningful considerations on input data (signals in our case) finally leading to some test case.

The same idea can also be used for automatic test generation. We outline it briefly on some example. So let us assume we have to generate a test executing branch D7, D3, D4, D5, D6, D7 (in procedure Delivmes in process Receiver). So for a moment we assume the process Receiver with its procedures to form a separate system (with corresponding declarations updated). So the signal MDTind is an ENV signal for the modified system, its parameters are treated as input values.

Now we try to find a feasible path containing the branch D7, D3,... . As one process is in fact a sequential program, heuristic methods from [21] can be applied. The shortest path containing the branch is R1, R2, R3, R4, R5, R6, R7, R8, R11, R12, R13, D1, D2, D3, D4, D5, D6, D7, D3,... , however, this path occurs to be infeasible (during the symbolic execution the solver founds the path condition contradictory). So by some (not described here) reasonable heuristic the next (by length) path is found, namely,
R1,R2,R3,R4,R5,R6,R7,R8,R11,R12,R13,D1,D2,D8,D9,R14,R4,R5,
R6,R7,R8,R11,R12,R13,D1,D2,D3,D4,D5,D6,D7,D3,D4,D5,D6,D7,...
The symbolic execution of the path gives the following path

condition (no timing conditions included, as there are no timers)

$MDTind_1^3 = dcheck(MDTind_1^1, MDTind_1^2)$

$1 \le MDTind_1^1 \; \& \; MDTind_1^1 \le 3$

$\neg(MDTind_1^1 = 1)$

$MDTind_2^3 = dcheck(MDTind_2^1, MDTind_2^2)$

$1 \le MDTind_2^1 \; \& \; MDTind_2^1 \le 3$

extract(modify((.false,false,false.),$MDTind_1^1$mod3,true),
$MDTind_2^1$mod3)=false

$MDTind_2^1 = 1$

extract(modify(modify(modify((.false,false,false.),
$MDTind_1^1$mod3,true),$MDTind_2^1$mod3,true),1,false),2)=true

The solver is able to find from the path condition unique values for numeric parameters of signals: $MDTind_1^1=2$, $MDTind_2^1=1$. The values of two other parameters are bound only by the conditions

$MDTind_1^3 = dcheck(MDTind_1^1, MDTind_1^2)$ (*)

$MDTind_2^3 = dcheck(MDTind_2^1, MDTind_2^2)$

These two conditions can be treated as preconditions for the process Receiver. Thus our solver can be extended so that it can be used not only for finding test values but also for generating preconditions from path conditions (we can also consider this process as a special kind of simplification).

Now we return to the whole system and find out (statically from declarations) that signal MDTind can come only from process MsgMan. Then we consider MsgMan alone in a similar manner with both MDTreq and MsgContr treated as ENV signals. However, this time the aim is different, namely, we have to find a path in MsgMan with the specified postcondition, namely, two instances of MDTind are sent with fixed values of the first parameter 2 and 1 respectively, in addition parameters are bound by (*). Using similar heuristics for path finding, the shortest path is found

M1,M2,M3,M4,M5,M6,M7,M3,M4,M5,M6,M7,M3,M8,M9,M10,M11,M12,
M13,M3,M8,M9,M10,M11,M12,M13,

which satisfies the given postcondition.

The postcondition and path condition from symbolic execution of the path together yield:

$MDTreq_1^1 = 2$

$MDTreq_2^1 = 1$

$MDTreq_1^3 = dcheck(MDTreq_1^1, MDTreq_1^2)$

$MDTreq_2^3$=dcheck($MDTreq_2^1$, $MDTreq_2^2$)

$MsgContr_1^1$=norm

$MsgContr_2^1$=norm

(see how postconditions are transformed by symbolic execution into preconditions). The order of ENV signals is

$MDTreq_1$,$MDTreq_2$,$MsgContr_1$,$MsgContr_2$ (timing again is unessential). As far as only the last two signals are true ENV signals from the system point of view, the search has to be continued to obtain two input signals MDTreq from Transmit with the first four equalities as postconditions. However similar analysis of Transmit yields the postcondition to be unfeasible - because under no circumstances sequence MDTreq(2,...) MDTreq(1,...) can issue from Transmit. So another path in MsgMan (with another postcondition arising for Transmit) must be found conforming with its own postcondition. The next (by length) path is the path induced by input signals $MDTreq_1$, $MDTreq_2$, $MsgContr_1$(Reord), $MsgContr_2$(Norm), $MsgContr_3$(Norm). This path gives the postcondition for Transmit with the first two equations modified

$MDTreq_1^1$=1

$MDTreq_2^1$=2

(and equations three and four remaining the same).

This is a completely "acceptable" postcondition for Transmit. The corresponding path is induced by (this time true) ENV signals $UDTreq_1$, $UDTreq_2$. So the complete sequence of ENV signals for the system is

$UDTreq_1$, $UDTreq_2$, $MsgContr_1$(Reord), $MsgContr_2$(Norm), $MsgContr_3$(Norm).

(or U, U, MR, MN, MN in terms of EST)

The complete ordering of signals is found substituting intermediate signals by ENV signal sequences generating these signals (likewise nonterminals are substituted by terminals in grammars). Timing conditions remain to be added to specify the test completely (the most stringent of them requesting that arrival times of all ENV signals are less than $UDTreq_1^T$+delta).

The search space for the method outlined is some tens of paths in the example considered (if powerful heuristics is used for path selection in one process). We also note that several branches in the "terminal" process (this time Receiver) can be searched for simultaneously, so reducing the complete search space for CTS even

more. So, similarly we can find that ENV signal sequence activating our "champion" branch R8,R9,R10 is (in terms of EST)

U, U, T, ML, MN, ML, MN

Maybe to find the latter path it would be more effective to consider at first Receiver alone and then Transmit and MsgMan together. Our estimate is that some hundreds of paths have to be considered to find CTS, a value completely acceptable for the example. So we hope a tool can be built using the approach outlined constructing CTS for pretty large protocols (and we hope also for large parts of electronic exchanges). However, such a tool would require some methods of reasoning on processes and pre/postconditions not completely formalized here.

We note only that the transformation of path postcondions to its preconditions by means of symbolic execution bears some resemblance to the methods used in program verification.

4 Conclusions

The results in both parts show that automatic test case generation has reached the status where practical implementations yielding acceptable results for programs of considerable size are possible. Certainly, such test generation systems would be complicated enough and will use the precise and heuristic methods described in the paper as well as some other ones. The main problem requiring some additional solutions is the path selection for traversing deeply "hidden" branches. In the theoretical approach the main principle used in path selection was state concept. Its modifications have proved their fitness also in a heuristic setting, however much work remains to be done to select appropriate heuristic state concepts for various classes of programs. One possible approach would be the attachment of formal comments by program authors to guide the automatic system in the right direction.

Acknowledgements

The authors would like to thank Prof. Jānis Bārzdiņš for the setting of the problem and valuable suggestions. They also wish to thank their colleagues at the Software Research and Development Department for help in the preparation of the paper.

References

1. Sauder R.L. General Test Data Generator for COBOL. - AFIPS Conference Proceedings, SJCC, 1962, pp. 317-323.

2. Hanford K.V. Automatic Generation of Test Cases. - IBM Systems Journal, 1970, vol. 9, No. 4, pp. 242-257.

3. Balzer R.M. EXDAMS - Extendable Debugging and Monitoring System. - In: Proc. 1969 SJCC, Montvale, N.Y., 1969, pp. 567-580.

4. Bārzdiņš J.M., Bičevskis J.J., Kalniņš A.A. Construction of Complete Sample Systems for Correctness Testing. - In: Mathematical Foundations of Computer Science, Berlin: Springer, 1975, pp. 1-12.

5. Howden W.E. Methodology for the Generation of Program Test Data. - IEEE Trans. Comput., vol C-24, pp. 554-559.

6. Clarke L.A. A System to Generate Test Data and Symbolically Execute Programs. - IEEE Trans. Software Eng., 1976, vol. SE-2, No. 3, pp. 215-222.

7. King J.C. Symbolic Execution and Program Testing. - CACM, 1976, vol. 19, No. 7, pp. 385-394.

8. Ramamoorthy C.V., Ho S.B.F., Chen W.T. On the Automated Generation of Program Test Data. - IEEE Trans. Software Eng., 1976, vol. SE-2, No. 4, pp. 293-300.

9. Pravilschikov P.A. Test Generation for Programs. - Avtomatika i Telemekhanika, 1977, No. 5, pp. 147-160 (In Russian).

10. Bičevskis J., Borzovs J., Straujums U., Zariņš A., Miller E.F. Jr. SMOTL - a System to Construct Samples for Data Processing Program Debugging. - IEEE Trans. Software Eng., 1979, vol. SE-5, No. 1, pp. 60-66.

11. Pozin B.A. A Method of Structural Test Generation for Programs. - Programmirovanie, 1980, No. 2, pp. 62-69 (In Russian).

12. Hogrefe D. Automatic Generation of Test Cases from SDL

Specifications. - In: SDL Newsletter, 1988, No. 12, pp. 34-52.

13. Kristoffersen F. Conformance Testing Based on SDL Specifications. - In: SDL'89: The Language at Work, North-Holland, 1989, pp. 257-266.

14. Bromstrup L., Hogrefe D. TESDL - Experience with Generating Test Cases from SDL Specifications. - In: SDL'89: The Language at Work, North-Holland, 1989, pp. 267-280.

15. CCITT : Specification and Description Language (SDL). Recommendations Z.100. - CCITT Blue Book, 1988, 199 p.

16. Saracco R., Smith J.R.W., Reed R. Telecommunication Systems Engineering Using SDL. - North-Holland, 1989, 633 p.

17. Auziņš A., Bārzdiņš J., Bičevskis J., Čerāns K., Kalniņš A. Automatic Construction of Test Sets: Theoretical Approach. - This volume.

18. Wirth N. Systematic Programming. - Prentice-Hall, 1973.

19. Hoare C.A.R. Algorithms 65; FIND. - CACM, 1961, vol 4, No. 1, p. 321.

20. Hoare C.A.R. Proof of Programm FIND. - CACM, 1971, vol. 14, No. 1, pp. 39-45.

21. Borzovs J.V., Urtāns G.B., Shimarov V.A. Program Path Selection for Test Generation. - Upravlayuschie Sistemi i Mashini, 1989, No. 6, pp. 29-36 (In Russian).

22. Borzovs J.V., Medvedis I.E., Urtans G.B. The Segment Method for the Solution of Systems of Equalities and Inequalities at Test Generation for Program Validation. - Upravlayuschie Sistemi i Mashini, 1990, No. 2, pp. 49-58 (In Russian).

23. Huet G., Oppen D. Equations and Rewrite Rules: a Survey. - In: Formal Languages: Perspectives and Open Problems, Academic Press, N.Y., 1980.

24. Futatsugi K., Goguen J.A., Jouannaud J.P., Meseguer J. Principles of OBJ'2. - In: Proceedings of Principles of Programming Languages, ACM, 1985.

25. Guidelines for the Application of Estelle, Lotos and SDL, Draft Manual. - CCITT, Geneva, 1988, 347 p.

26. Stenning N.V. A Data Transfer Protocol. - Computer Networks, 1976, No. 1, pp. 99-110.

27. Bergstra J.A., Heering J., Klint P. (ed.) Algebraic Specification. - ACM Press, N.Y., 1989, 397 p..

28. Sato F., Katseryama K., Mizuno T. TENT: Test Sequence Generation Tool for Communication Systems. - In: FORTE'89, Proceedings of 2nd Int. Conf. on Formal Description Techniques, North Holland, 1990, pp. 1-6.

29. Chan W.Y.L., Vuong S.T., Ito M.R. On Test Sequence Generation for Protocols. - In: Proceedings of the IFIP WG 6.1 Nineth Int. Workshop on Protocol Specification, Testing and Verification, 1989, North Holland, 1989.

AGGREGATE APPROACH FOR SPECIFICATION, VALIDATION, SIMULATION AND IMPLEMENTATION OF COMPUTER NETWORK PROTOCOLS

Henrikas Pranevitchius

Kaunas University of Technology
Faculty of Informatics
V.Juro 50, Kaunas, 2330028
Lithuania

ABSTRACT

The application of aggregate approach for the formal description, validation, simulation and implementation of computer networks protocols is considered in the paper. With this approach the above mentioned design stages can be executed using a single mathematical scheme. The method of reachability states is used for the validation of protocol general properties, while individual characteristics are analysed by the invariant method which enables to verify the correctness of the invariant by protocol formal description. Aggregative mathematical schemes are used in the specification languages AGREGAT-84 and ESTELLE/AG applied in creating protocol analysing systems simulation and validation of protocols. Protocol automated implementation method based on the specification language ESTELLE/AG is presented. Formal description and results of alternating -bit protocol validation and simulation as its speciffication in AGREGAT 84 and Estelle/Ag are presented for illustration.

Introduction

The main function of computer networks software is providing interaction of information processes realized in a distributed and, as a rule, non - homogenous mediumm. The main part of this software, namely, detailed system agreement and rules of interaction, realized in computer networks is called the protocol.

SDL '91
EVOLVING METHODS

Proceedings of the Fifth SDL Forum
Glasgow, Scotland, UK, 29 September – 4 October, 1991

edited by

Ove FÆRGEMAND
TFL
Hørsholm, Denmark

Rick REED
GPT
Coventry, UK

RReed

Most pleased to have you on the committee and welcome you to the U.K.

1991

NORTH-HOLLAND
AMSTERDAM • NEW YORK • OXFORD • TOKYO

ever, according to [7], service features in the telephone exchange systems should be independent of each other, so they can be tested separately or in small portions.

Evidently, all the aforementioned applies only to the construction of test cases with 100% coverage according to criterion C1. If, say, 90% coverage is satisfactory, then the generation of tests will speed-up several times.

7. ACKNOWLEDGMENTS

The authors would like to thank Prof. Jānis Bārzdiņš and Prof. Audris Kalniņš for the setting of the problem and valuable suggestions.

8. REFERENCES

1 Hogrefe D. Automatic Generation of Test Cases from SDL Specifications. - In: SDL Newsletter, 1988, No. 12, pp. 34- 52.

2 Bromstrup L., Hogrefe D. TESDL - Experience with Generating Test Cases from SDL Specifications. - In: SDL'89: The Language at Work, North-Holland, 1989, pp. 267-280.

3 Sato F., Katseryama K., Mizuno T. TENT: Test Sequence Generation Tool for Communication Systems. - In: FORTE'89, Proceedings of 2nd Int. Conf. on Formal Description Techniques, North-Holland, 1990, pp. 1-6.

4 Bourgnet-Rouger A., Combes P. Exhaustive Validation and Test Generation in ELVIS. - In: SDL'89: The Language at Work, North-Holland, 1989, pp. 231-245.

5 Holzmann G.J. Automated Protocol Validation in ARGOS: Assertion Proving and Scatter Searching. - IEEE Trans. on Software Eng., vol. SE-13, No. 6, 1987.

6 Kalnins A. Global State Based Automatic Test Generation for SDL. - This volume.

7 CEPT Handbook on Services and Facilities Offered to the Subscribers in Modern Telephone Systems. - CEPT, 1984.

8 LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. - ISO DIS 8807, 1987.

9 Kroger F. Temporal Logic of Programs. - Springer Verlag, 1987, 148 pp.

10 Barzdin J., Bicevskis J., Kalnins A. Construction of Complete Sample Systems for Correctness Testing. - In: Mathematical Foundations of Computer Science, Springer Verlag, 1975, pp. 1-12.

11 Clarke L.A. A System to Generate Test Data and Symbolically Execute Programs. - IEEE Trans. Software Eng., 1976, vol. SE-2, No. 3, pp. 215-222.

12 Barzdin J., Kalnins A., Auguston M. SDL Tools for Rapid Prototyping and Testing. - In: SDL'89: The Language at Work, North-Holland, 1989, pp. 127-133.

SDL '91: Evolving Methods
O. Faergemand and R. Reed (Editors)
© Elsevier Science Publishers B.V. (North-Holland), 1991

Global State Based Automatic Test Generation for SDL

A.Kalniņš

Institute of Mathematics and Computer Science
University of Latvia
Raiņa Bulv. 29, Riga 226250, Latvia

Abstract

The possibility of automatic test case generation for SDL systems is considered. Methods for optimal global state definition including not only process states but also essential information on variable values are described. Based on these methods an algorithm and its implementation for automatic test set generation for a class of protocol programs is described. Test generation experience on sliding window example is shown.

1. INTRODUCTION

Automatic test case generation for ordinary sequential programs first became popular in mid seventies [1,2]. A significant contribution in this area was made by research group headed by prof. Barzdin at Latvia University [3,4,5]. While automatic test generation hasn't become an essential part of general software engineering, it has appeared to be very significant for telecommunications systems described in SDL and other related languages. Both theoretical and practical aspects of automatic test generation experience by Latvia University group (including the author of this paper) recently have been reported in [6,7], including also some SDL oriented approach.

The main idea of this paper is to carry over the experience obtained in automatic test generation for sequential programs to SDL, especially the state concept.

In recent years the automatic test generation for SDL has gained significant popularity, specially in protocol area [8,9]. This is due to large interest by practitioners, particularly in conformance testing. Up to now SDL systems are mainly considered as collections of finite state machines. So global state of the system is considered in general as a tuple of all process states and internal queue contents. As states in SDL can be coded by finite set valued variables and vice versa, at least some variables must be considered as part of the global state. This paper focuses on the matter how to include the information on the variables really influencing the behaviour (essential variables) in the global state while keeping the state space as small as possible.

The other issue is test coverage criteria. It is assumed quite frequently that all global states must be exercised thus leading to a state explosion. In this paper more attention is paid to a "program-like" criterion C1 - to execute all branches in the system, clearly reducing

the search space. So the aim of the paper is to present a method yielding thorough automatic testing for realistic systems on not very powerful computers.

There exist some similar approaches [10,11,12]. The closest is the PROTAN approach [10] for systems described in Estelle. The main difference is that [10] gives no method to decide what information on variables should be included in the global state. It is also focused on reachability analysis, not on test generation.

2. TESTING GOALS AND CRITERIA

It is very common to consider [8,10] that the testing of an SDL system is complete if every global state (as a tuple of all process states) is reached. While it is reasonable in reachability analysis for deadlock detection, it is not always so for ordinary testing.

So we propose to carry over the completeness criterion C1, widely used for sequential programs, also to SDL. We say that a test set T (i.e., a set of external signal sequences) is complete for the given system S with respect to C1, if every executable branch in every process (procedure) of S is executed at least once on some test of T. By a branch we understand both input branch in a state and ordinary decision branch.

Testing in SDL can have two different purposes. On the one hand, SDL specification itself should be tested as a program with manual checking of results. In this case criterion C1 seems to be more appropriate. On the other hand, in conformance testing the SDL specification is used as a reference model for an implementation of the protocol unit, in this case much larger test sets are recommended.

The methods for practical test generation described in this paper can be used with various criteria, only the termination conditions must be adapted. Certainly, the used resources heavily depend on the criterion.

3. SUBSETS OF SDL WHERE AUTOMATIC TEST GENERATION IS POSSIBLE

In [6] various sequential programing languages are described where completely automatic (algorithmic) test generation is possible (or proven to be impossible). We carry over most of these results to subsets of SDL.

So we consider a subset of SDL with only simple predefined types (integer, character, boolean) and statements including signal sending/receiving (with parameters, both internal and to/from environment), simple assignments of type x1:=x2, x1:=c, decisions of type x1<x2 (other relational operators and constants also allowed), other statements not allowed.

The following list of results has been obtained:

1. For one process there is an algorithm generating test sets according to C1 (the process is communicating with the environment only)

2. There is no algorithm generating test sets if two communicating processes and save statement are allowed

3. n processes allowed, but no save - an algorithm seems to exist (not proven up to the moment)

4. n processes, save allowed, but there is a constant N such that any queue length never exceeds N - algorithm exists (but the case itself cannot be deduced formally from syntax restrictions)

5. One process, with one-way counter z added (with statements z:=c, z:=z+1, z<x) - no algorithm exists

6. One process, with n timers and save added - algorithm exists (this result has been obtained by K.Cerans already in SDL terms in [6]).

The above mentioned results show that in general completely automatic test generation is possible only for quite narrow subsets of SDL. Nevertheless the methods used to obtain the results, especially the notion of the global state, can be used in practice for much wider classes of SDL systems.

So in the next sections we consider in general a large subset of SDL with only create, import/export, view/reveal facilities execluded. Only specific state construction methods are related to SDL subsets described in this section. There is also a semantic restriction that no two statements in the whole system are executed simultaneously, so the execution history of a system can be described uniquely by a sequence of events (path).

4. GENERAL PRINCIPLES OF GLOBAL STATE

All the results in the previous section, where algorithmic construction of test sets is possible, are based on an appropriate global state concept.

The global state is some condensed information about an SDL system during its execution, i.e., after a given sequence of statements W has been executed. The information coded in the state should be rich enough to determine which future actions in the system are legal.

Let us denote the state after execution of the sequence W by S(W). State concept is said to be *correct* if from the fact that two execution sequences W_1 and W_2 in a system lead to equal states, i.e., $S(W_1)=S(W_2)$ there follows that both W_1 and W_2 have the same set of possible continuation sequences. Only reachable states (emerging after some legal execution sequence in the system) are considered.

At the same time, in order to make the test construction algorithmic, the set of possible states for every SDL system in the class has to be finite.

If there is a correct concept of state for the given class of SDL systems yielding finite sets of states, then the set of tests for the given system M can be built the following natural way, using the reachability graph [6]. We start with an empty initial state S_0 (and mark it as the "root" of the graph). Then we find a statement L_1 in one of the processes of M which can be executed in the given situation. So we construct the state $S(L_1)$ and add a new vertex $S(L_1)$, connected by an edge to S_0. Now we have two vertices from which to continue the process the same way. If a state S is obtained which already exists in the graph, no new vertex is added, but an edge is drawn to the existing instance.

Let us give a small example (see Fig. 1). Let a system consist of two processes P1 and P2.

The state concepts for various system classes differ in the way state S(W) is generated from the execution sequence W. We assume the example to belong to the solvable case 4,

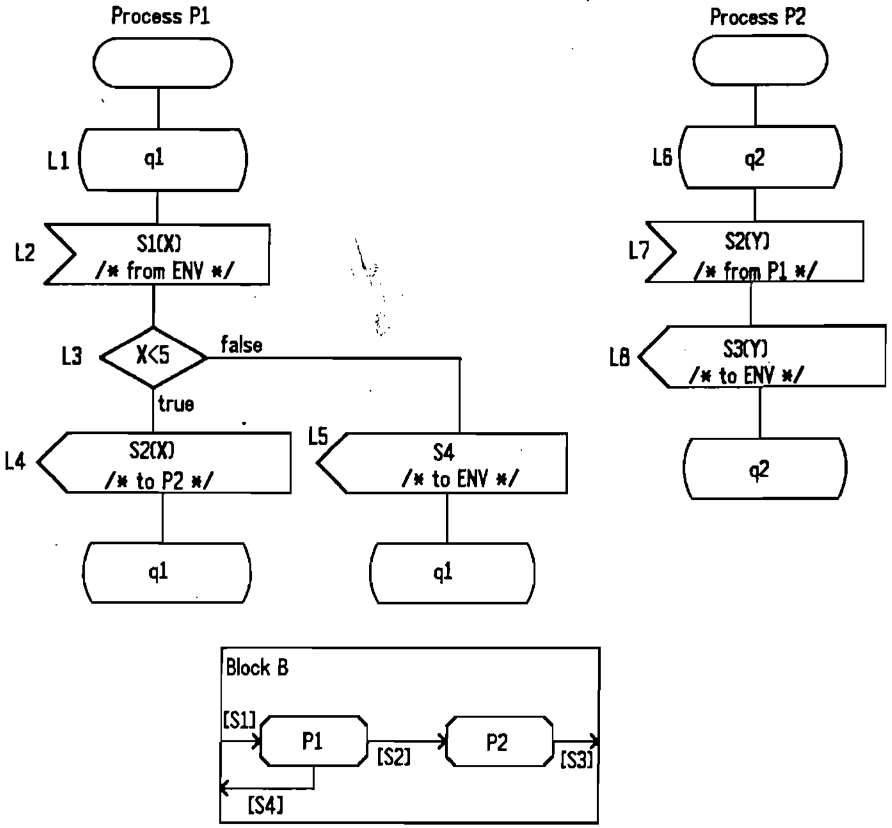


Fig. 1. Example of a system

so the state will contain equalities and inequalities between process variables and (bounded) queue contents.

Initially P1 and P2 are in states q1 and q2 respectively. The first possible event in the system is consumption of signal S1 from environment by P1 (statement L2). The state contains only the statement labels L2;L6. The only next event is execution of L3 by P1 with two possible exits, exit true is assumed, yielding the state L3(t):X<5;L6. We stress once more that the state does not contain the value of X, only the information on admissible values of X. After sending the signal to P2 the state is L4:X<5; L6:queue(P2)=<S2(X)>. Now the choice of continuations is possible, one of them is signal consumption by P2: L4:X<5; L7:Y=X. Fig 2. shows the initial fragment of the reachability graph.

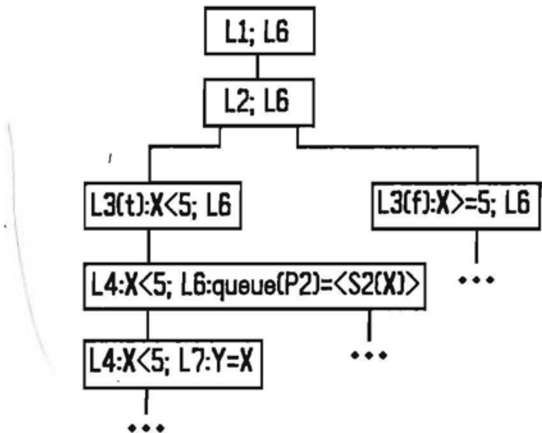


Fig. 2. Initial fragment of the reachability graph

When more than one process can be active, all possible "schedulings" of actions should be shown in the graph. It should be noted that in some cases inequalities are generated in a more sophisticated manner.

To find a test a finite path in the reachability graph is taken. A system of equalities and inequalities with respect to parameters of input signals is formed and solved. The solution should exist by construction of state (as states are built only for executable paths, executability fact actually is coded in the state itself). In the example considered the only inequality for the leftmost path in Fig. 2 implies that the parameter of S2 must be less than 5. Some set of paths covering the graph (in the sense of the selected criterion, in our case C1) yields the complete test set.

5. METHODS FOR GLOBAL STATE REDUCTION

The number of global states in the reachability graph is quite large even for small programs. To reduce the size of this graph we introduce two notions: *essentially located statements* (ELS) and *essential variables* (EV). For sequential programs these notions were introduced in [4], see also [6].

The set of ELS for an SDL system is a set containing at least one statement from each loop in every process. The start and stop statements are also considered as ELS. The loops in SDL programs are formed by both conventional control statements and state - nextstate pairs, in the latter case, namely, the state is recommended as ELS. The main idea behind the notion of ELS is that there is only a finite number of paths between ELS, there is no infinite path without ELS.

For every ELS a set of EV is defined. A variable v is said to be essential for a certain ELS if there is an execution sequence beginning with the ELS such that the value possessed by the variable v immediately after ELS is used in some branching statement of the path.

The use of the value means that either the variable v appears in some decision statement, e.g., $v<9$, (before a new assignment to v) or v is assigned to some other variable u which, in turn, is used in branching. The values could even be transferred by means of a signal to another process and used in branching there.

There exists a formal algorithm (based on static analysis of process diagrams) to decide whether the variable is essential for the given ELS. The version of this algorithm for sequential programs is described in [6].

It should be noted that implicit SDL variables, i.e., signal queues and active timer sets are EV in general.

With ELS and EV the size of the reachability graph can be reduced significantly (while preserving the correctness of state concept). Now only the statement label tuples containing one ELS from each of the processes are taken as vertices, only the information about EV is coded in the global state. The set of tests is obtained from the graph in a similar (though more sophisticated) way.

The reduction of state space is significant. So in the previous example the only ELS are states q1 and q2, no variable is essential except signal queues, the reachability graph becomes trivial (consisting of one vertex).

The construction of the reachability graph is based on some kind of symbolic execution of an SDL system (the notion is widely used for sequential programs [2,13] and defined accurately for SDL by the author of this paper in [7]). Symbolic execution yields for the given path the symbolic values of variables and path condition, a system of equalities and inequalities expressing the condition under which the path can be executed. The information contained in the state is obtained (according to the selected state concept) from symbolic values of essential variables and from path condition.

6. FURTHER HEURISTIC METHODS FOR STATE REDUCTION

The selection of ELS may be restricted even more while still preserving the correctness of state concept. We must keep in mind that the only requirement is that the length of every path from one ELS to other must be bounded.

Further assumptions are valid only for a smaller subset of SDL (maybe specified not completely formally). The subset is characterized by the following properties:

- parameters of external signals (i.e., signals sent from environment) are not essential (i.e., they are not used directly or indirectly in decisions)
- the only variables which are essential are either of counter type (i.e., using statements like $v:=c$, $v:=v+c$, $v:=v-c$, $v1:=v2$, $v1<v2$, $v1=v2$) or variables with bounded set of values (like boolean, enumerated types etc.).

There are no restrictions on operations with nonessential variables.

Nearly all popular protocol specifications belong to this class. Certainly, the first restriction to be valid, protocol entities should be selected properly, in general, the complete description of a protocol layer is to be taken.

It can be shown that for this class of SDL systems single loops of the form for $i:=v1$ to $v2$ (actually expressed by normal SDL statements) cannot cause unbounded paths. So no ELS must be taken from such loops, only loop bounds (if they are not constants) are to be taken as EV. For nested loops the abovementioned is not always true.

Likewise, frequently ELS are not necessary in loops caused by state/nextstate. It is so in the case, when only "internal" signals are received in the state, and signals in the state are not received from a process, to which signals are sent in the transitions of the state (i.e., there is no internal "signal loop" between two or more processes, not involving any external signal).

The abovementioned heuristic methods (and some similar) allow for many protocol specifications to take only states where external signals are received as ELS. The finding of EV is as described before, but their values must simply be recorded in the state (because there is no more economic coding for counters). In general, signal queues are EV, for they influence signal branch execution. But in many cases a simplifying semantic assumption can be made that internal signals have priority over the external ones, i.e., no external signal is received until all internal queues (certainly, not saved ones) are empty. In this case queues for processes where states are not ELS and no saves are contained are not EV and must not be kept in global state.

So the global state space is not very large even for relatively long paths thus allowing to generate tests for real protocol specifications. On the other hand, such a choice of ELS is very convenient for implementation.

In the described heuristic approach the state correctness is preserved. At the same time the finiteness of state set can be no more guaranteed, some other heuristic methods to terminate the reachability graph construction must be used.

Let us note also that the nature of essential variables for the described class of SDL systems makes the symbolic execution normally used in the state construction coincide with the actual execution.

Additional heuristic idea to be used to reduce the state space is not to save completely the values of essential variables v1,v2,..., but try to find functions f1,f2,..., such that state with fi(vi) instead of vi is also correct and value space of fi is not large. Certainly, there is no general method for finding such functions.

7. IMPLEMENTATION OF AUTOMATIC TEST GENERATION

7.1. Implementation environment

The described methods for global state based automatic test case generation for SDL systems are being implemented in RIGA-SDL integrated environment.

The earlier version of RIGA-SDL has been described in [14]. The current version is implemented on IBM PC and has been in industrial use for nearly a year. The SDL version used is SDL-88 [15] with the following differences. Pascal data types are used instead of abstract data types, structuring facilities like subprocesses, block arrays, explicit loop statements are added, some rarely used options like import/export are excluded. RIGA-SDL has a unified menu based user interface. Tool components include graphical editors both for blocks and processes, SDL to Pascal compiler with complete "graphic" error diagnostics, runtime kernel implemented in Pascal and supporting complete SDL semantics (in simulated time mode). The debugging facilities include online/offline signal and process activation trace analyzer and sequence chart generator. Testing environment contains facilities for convenient external signal input both in interactive mode and from

test case files and for test result recording. The environment is specially oriented towards large scale SDL specification testing and debugging.

7.2. Implementation methods

The automatic test case generator is based on the existing SDL compiler, all the necessary information about SDL system is extracted from its intermediate representation. The test generation facilities in fact will consist of two parts. One is static analyzer determining ELS and EV. The other is global state graph generator and test case extractor. For the moment only the second part has been developed, ELS and EV must be determined manually. ELS which are states with possible external signal input are registered automatically. The values of determined EV are either registered as they are, or by means of user supplied coding functions.

To find the global state after some external input, the normal compiler generated code is used. Only a modified runtime kernel is used which stops the execution at the next ELS and passes control to state generator. It also supplies on the request from the generator the values of EV and other relevant information (process states, queue contents, active timer sets). The state generator forms the current global state, compares it with the saved state list and updates the list accordingly. The current branch of the state graph (tree) is also recorded. The next external signal to be exercised is found by means of some heuristics and passed to the execution system (it may also be a "time delay" instruction to let a timer ring).

The automatic termination condition of state space exhaustion is not allways effective as state space can be infinite. Termination condition "the completeness criterion C1 has been reached" is also added. User supplied termination conditions can also be used both in the form of a bound on search depth and a "cut-off" function for states with too large variable values.

The last phase of the test generation analyzes the constructed graph and generates test cases (external signal sequences with time marks). Some sort of test case minimization is included. For the moment tests are generated as input files for RIGA-SDL testing environment, but TTCN [8] notation could be used as well.

8. EXPERIENCES IN AUTOMATIC TEST GENERATION

As the development of the test generator has been recently finished, only some examples have been examined. One of them is the popular sliding window protocol [16,17,7]. The description given in [17] is slightly transformed to adapt it to Pascal data types used in RIGA-SDL. The protocol description by the transmitter and receiver processes is not changed, both window sizes are set to 3 for test runs. The medium description (with two processes) is made completely deterministic by introducing external control signals determining medium impact on the current message/acknowledgement. So the set of external signals contain UDTreq, MsgNorm, MsgLose, MsgCorrupt, MsgReorder, AckNorm, AckLose, AckCorrupt, AckReorder (plus the possibility for a timeout to occur). The protocol is completely in the described class. EV for the sliding window contain the basic counters (HighestSent, LowestUnacked, NextRequired) , reception tags for receiver window (booleans $AlreadyReceived_i$) and the explicit message and acknowledgement queues in

medium processes (with a user specified coding function). For criterion C1, with a user specified cut-off rule for states added, 760 states have been constructed and 29 test cases have been generated (with the maximum length of 7 signals). The test generation requires 40 minutes on 12 MHZ PC AT for the sliding window example, the performance is expected to be improved after fine-tuning of the test generator. To compare the results, we have implemented also algorithm from [8], where only process states and queues are included in the global state. Much more states and test cases were generated, but not all system branches were traversed. This is due to the fact that signal sequences of length 7 are to be examined to reach C1 (with maximum branching at any point 10).

The other example examined is Kermit protocol, where a C1-complete test set can also be constructed.

9. CONCLUSIONS

The experiments done so far have shown that automatic test case generation based on fine tuned correct global state concept is quite acceptable in practice. The state sets necessary to reach criterion C1 seems to be by order of magnitude less than those for reachability analysis. Certainly, the testing goals are a bit different but program-like testing is necessary also for protocols, moreover, some kinds of deadlocks can be fixed by our methods during state graph construction.

Of course, many improvements can still be applied to the state generation described here. The other future direction is to apply a proper symbolic execution during global state construction. So a much wider class of SDL systems including telephone switches could be covered. This approach (briefly sketched in [7]) requires powerful symbolic expression simplification (possibly based on term rewriting systems). An alternative approach to test generation for telephone systems is given in the other paper by Latvia University group [18].

10. REFERENCES

1 Howden W.E. Methodology for the Generation of Program Test Data. - IEEE Trans. Comput., vol C-24, pp. 554-559.
2 Clarke L.A. A System to Generate Test Data and Symbolically Execute Programs. - IEEE Trans. Software Eng., 1976, vol. SE-2, No. 3, pp. 215-222.
3 Barzdins J.M., Bicevskis J.J., Kalnins A.A. Construction of Complete Sample Systems for Correctness Testing. - In: Mathematical Foundations of Computer Science, LNCS, Vol. 32, Springer-Verlag, 1975, pp. 1-12.
4 Bicevskis J., Borzovs J., Straujums U., Zarins A., Miller E.F. Jr. SMOTL - a System to Construct Samples for Data Processing Program Debugging. - IEEE Trans. Software Eng., 1979, vol. SE-5, No. 1, pp. 60-66.
5 Barzdins J.M., Bicevskis J.J., Kalnins A.A. Automatic Construction of Complete Sample Systems for Program testing. - In: Proc. IFIP Congress, 1977, North-Holland, 1977, pp. 57-62.

6 Auzins A., Barzdins J., Bicevskis J., Cerans K., Kalnins A. Automatic Construction of Test Sets: Theoretical Approach. - In: LNCS, Vol 502, Springer-Verlag, 1991, pp. 287-360.
7 Borzovs J., Kalnins A., Medvedis I. Automatic Construction of Test Sets: Practical Approach. - In: LNCS, Vol 502, Springer-Verlag, 1991, pp. 361-433.
8 Bromstrup L., Hogrefe D. TESDL - Experience with Generating Test Cases from SDL Specifications. - In: SDL'89: The Language at Work, North-Holland, 1989, pp. 267-280.
9 Kristoffersen F. Conformance Testing Based on SDL Specifications. - In: SDL'89: The Language at Work, North-Holland, 1989, pp. 257-266.
10 Tienari M., Aaltonen K., Eloranta J., Keskinen J., Lehtinen K., Summanen L., Tapila K., Turunen I. PROTAN 88 - a Software tool for Verifying Communication Protocols Specified with an Extended State Transition Model. - University of Helsinki, Dept. of Comp. Sci., Report No. A - 1988 - 5, Helsinki, 1988, 65 p.
11 Sato F., Katseryama K., Mizuno T. TENT: Test Sequence Generation Tool for Communication Systems. - In: FORTE'89, Proceedings of 2nd Int. Conf. on Formal Description Techniques, North Holland, 1990, pp. 1-6.
12 Chan W.Y.L., Vuong S.T., Ito M.R. On Test Sequence Generation for Protocols. - In: Proceedings of the IFIP WG 6.1 Ninth Int. Workshop on Protocol Specification, Testing and Verification, 1989, North Holland, 1989.
13 King J.C. Symbolic Execution and Program Testing. - CACM, 1976, vol. 19, No. 7, pp. 385-394.
14 Barzdin J.M., Kalnins A.A., Auguston M.I. SDL Tools for Rapid Prototyping and Testing. - In: SDL'89: The Language at Work, North-Holland, 1989, pp. 127-134.
15 CCITT: Specification and Description Language (SDL). Recommendations Z.100. - CCITT Blue Book, Volume X Fascicle X.1, 1990.
16 Stenning N.V. A Data Transfer Protocol. - Computer Networks, 1976, No. 1, pp. 99-110.
17 Guidelines for the Application of Estelle, Lotos and SDL, Draft Manual. - CCITT, Geneva, 1988, 347 p.
18 Grasmanis M., Medvedis I. Approach to Behaviour Specification and Automated Test Generation for Telephone Exchange Systems. - this volume.

SDL '91: Evolving Methods
O. Faergemand and R. Reed (Editors)
© Elsevier Science Publishers B.V. (North-Holland), 1991

TEST SELECTION BASED ON SDL SPECIFICATIONS WITH SAVE

Gang LUO, Anindya DAS and Gregor v. BOCHMANN

Department d'IRO, Universite de Montreal, C.P. 6128,Succ.A, Montreal, P.Q., H3C 3J7, Canada. E-mail:luo@iro.umontreal.ca, Fax: (514) 343-2155 .

Abstract
The signal SAVE function is one of the characteristics distinguishing SDL from conventional high-level specification and programming languages. However, this feature increases the difficulties of testing SDL-specified software. This paper proposes a method for developing tests for system testing based on SDL specifications including the SAVE construct. It also investigates the effects of the input queue of SDL.

1. INTRODUCTION

During the development of SDL, the first feature added to SDL which considerably increased the difficulty of transforming SDL to CHILL was the SAVE construct[1]. However, the SAVE function increases SDL's descriptive power considerably by providing a concise formalism for expressing the indeterminate order of arrivals of input signals. Its presence raises a challenge in testing SDL-specified software. Some initial efforts have been made to tackle this issue [2,3]; a formal method was proposed in [2] and a similar framework was introduced informally through examples in [3]. However they did not address the case where the SAVE construct has several SIGNALs, a case which is quite common.

This paper investigates software testing based on SDL specifications when SAVE constructs contain several signals. Our approach is to transform an SDL description containing SAVE to an equivalent SDL description without SAVE which preserves the same relationship between input signal sequences and output signal sequences. The testing methods for the finite state machine can then be applied [4,5,6,7]. In the case of an SDL description which does not have an equivalent finite state machine (FSM) without SAVE, an alternate approach is proposed.

Our approach assumes that the SDL description is a FSM with the SAVE extension. Such a description can be obtained from a general SDL specification in the following fashion. The variable extension can be eliminated by transforming conditions which cause branches at the DECISION construct; the combinations of inputs and conditions can be used to create a FSM with new inputs being the combination of conditions and original inputs. The details can be obtained from [3]. By ignoring parameters and other variables, we obtain a finite state machine containing SAVEs and an input queue, which we call an "*SDL-machine*".

The rest of the paper is organized as follows. Section 2 is devoted to the fault model and gives a brief introduction to the *SDL-machine* formalism. Section 3 investigates the relations between *SDL-machines* and FSMs in order to adopt the testing methods for FSMs to test *SDL-machines*. We propose an algorithm to transform an *SDL-machine* to an equivalent FSM which preserves the input/output relation. For the *SDL-machine* which cannot be transformed to an equivalent FSM leaving the input/output relation unchanged, another algorithm is given to transform it to a FSM which approximates the original *SDL-machine*. Section 4 handles the test case selection methods based on the results of section 3, and analyzes the test coverage thus obtained.

The 6th International Conference on Software Engineering and Knowledge Engineering

Co-Sponsored by
Knowledge Systems Institute
University of Latvia
University of Pittsburgh

In Cooperation with
IEEE Computer Society

Technical Program, June 21-23, 1994

JURMALA, LATVIA

GRADE Windows :an Integrated CASE Tool for Information System Development [*]

J.Bārzdiņš, A.Kalniņš, K.Podnieks, I.Etmane,
A.Auziņš, A.Kālis, P.Krastiņš, S.Rozenfelds
University of Latvia
Institute of Mathematics and Computer Science
Rainis boul. 29, Riga LV-1459, Latvia

Abstract

The paper outlines the basic ideas of unified specification language called GRAPES/4GL and corresponding toolset called GRADE Windows. The toolset is aimed to support all system development phases including analysis, requirements specification,design and implemntation.

1. Introduction

By integrated CASE tools (ICASE tools) we understand computer aided system and software engineering tools supporting all system development phases including analysis, requirements specification, design and implementation. It is a generally accepted view that complex software systems, including information systems, can be built only using ICASE tools. The core of such tools is a specification language on which all tool activities are based. High level specification languages accepted in practice have been developed for telecommunications systems (SDL, LOTOS, Estelle) and for process control applications (e.g., STATEMATE). On the basis of these languages the corresponding CASE tools have been developed (e.g., SDT[1], GEODE[2], SPECS[3], STATEMATE[4] et.al.).

However, in the area of information systems the situation is much worse. There is no generally accepted high level specification languages for this area. There exist specification languages with slightly theoretical bias (e.g., SPEC[5], SF[6]). However, it seems that up to the moment they have not reached the status of real industrial languages with full CASE tool support. In fact, there are specification language elements present in some popular CASE tools, e.g., IEF[7], yet the specification language has not obtained the status of independent and well-defined component in these tools.

In recent years the research and development in the area of specification languages and CASE tools has been significantly turned towards object orientation ([8], [9], [10].)

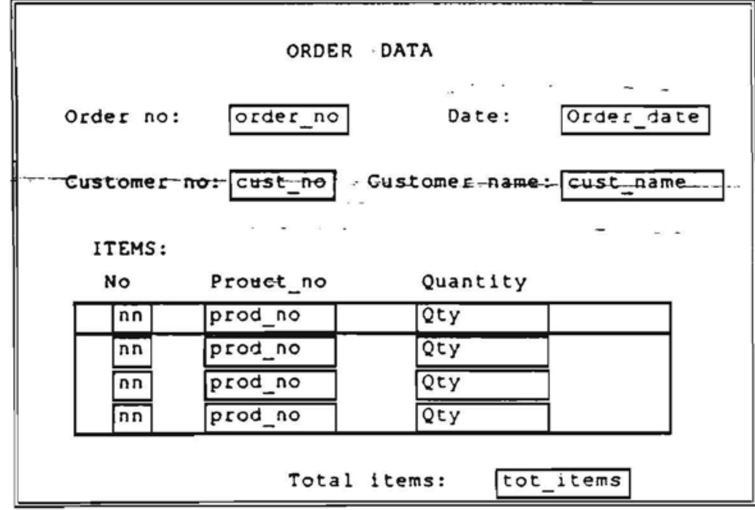
If we look at the situation some years before one of the practically most advanced languages for analysis and design of information systems was GRAPES-86 proposed by Siemens [11]. We started our research and development three years ago with a goal to develop further the GRAPES-86 language and to build the corresponding CASE tools. The research resulted in the development of the language GRAPES/4GL as an extension of GRAPES-86 and the corresponding CASE tools named GRADE Windows. This paper outlines the basic ideas of GRAPES/4GL language and GRADE Windows tools (short description of GRAPES/4GL and GRADE Windows is given also in [12]).

Several years ago, when we started our developments, there existed already advanced CASE tools like IEF [7] (and many others). However, a significant distinction of GRAPES approach was putting in the foreground the description of system structure and its external interfaces (i.e., what messages enter the system and what leave). In other words, GRAPES approach is focused on system understanding from the view point of incoming/outgoing information. At the same time IEF approach puts in the foreground the conceptual data model of the system and its databases, around which applications are then being built. To our mind, these are two principally different approaches each having its own merits and flaws.

2. Basic Ideas of Language GRAPES-86

As it was mentioned in introduction, the basis for GRAPES/4GL is Siemens Nixdorf system design language GRAPES-86 [11]. This language united well known diagramming techniques - data flow diagrams, ER-models, graphic process descriptions into a coherent

[*]This work was supported by Software House Riga and Infologistik GmbH, Munich




```
TYPE REFERENCE: item, order
SIZE: 15, 55
FIELD LIST:
order_no, REQUIRED
Order_date, TYPE DATE
cust_no
cust_name, NOENTRY
ARRAY Items, 4 ROWS
 nn = AUTONUM
 prod_no, REQUIRED
 Qty, REQUIRED
END ARRAY Items
```

Fig. 1

graphic design language. The main paradigm of the GRAPES language is a multilevel static system structuring, and description of system behaviour as a set of processes communicating via messages and performing each own job in response to received messages.

Formally, GRAPES-86 uses a fixed set of diagram types for system description :

CD for system structure description;
DD - for graphic datatype definition;
IT - for communication description;
ER- for Entity-relationship modeling (as a standalone feature);
PD for graphic process/procedure behaviour descriptions;
SD - for procedure/module interface definition
DT for process/procedure local variable definition.

A system in GRAPES-86 is described as a model representing a hierarchy of the abovementioned diagrams, the topmost one being a CD diagram. This diagram hierarchy is called a model tree.

However, GRAPES-86 lacks some important features for system design description and doesn't support actual implementation at all. The most outstanding deficiencies are lack of user interface description and missing data manipulation features.

3. Basic Ideas of Language GRAPES/4GL

GRAPES-86 is strongly oriented towards the needs of system analysis and "coarse" design level. As a result, if we follow the GRAPES-86 methodology, there is a large gap between high level specification and implementation in some target environment. The basic goal of GRAPES/4GL approach is to reduce this gap as far as possible. In order to achieve this goal we tried to design the GRAPES/4GL language so that it can serve both as specification and programming language. Hence, we had to incorporate into GRAPES typical features of advanced 4th generation languages, not "damaging" the language features oriented towards specification. Such a

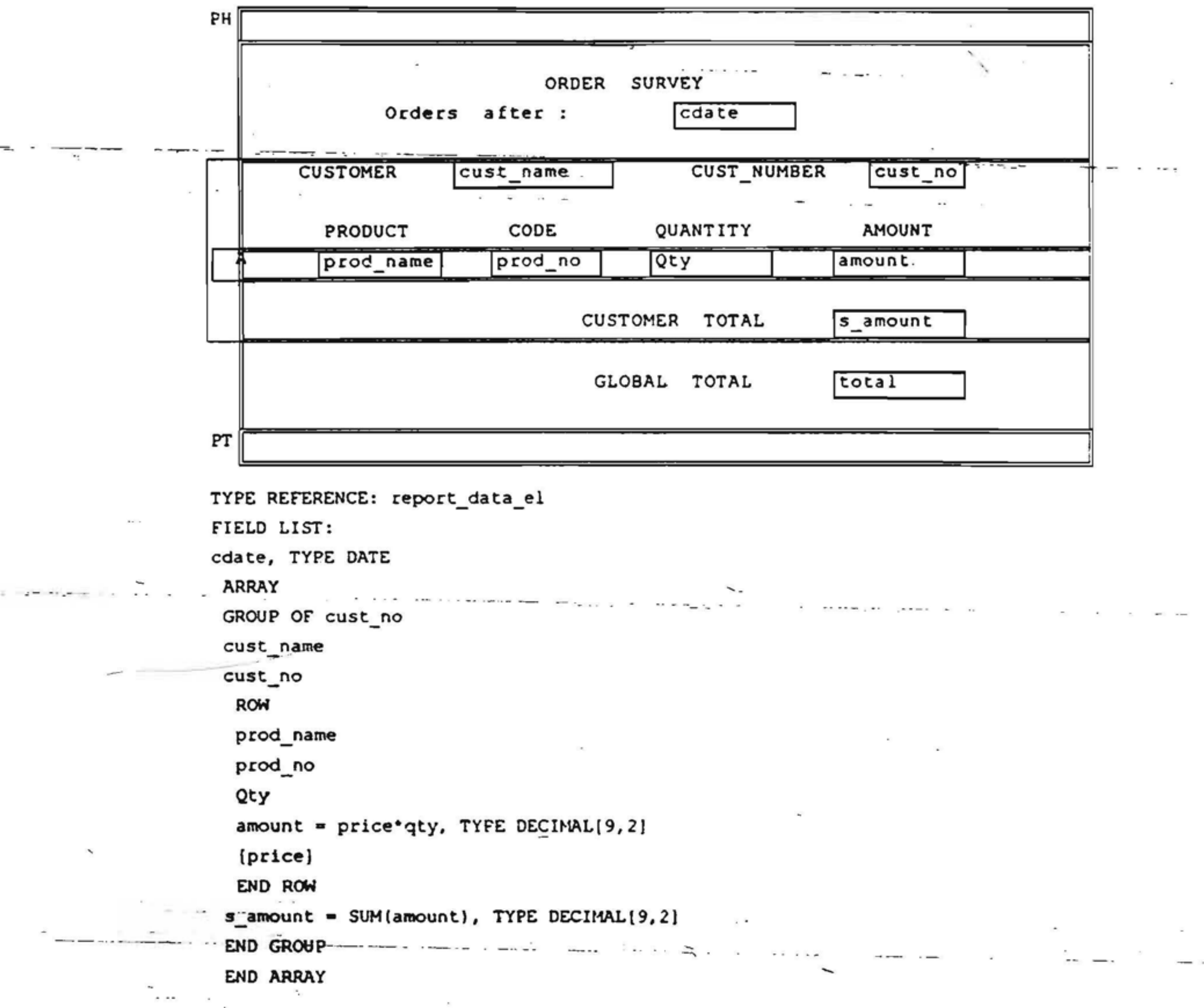


Fig. 2

unified language, valid for all development phases, from our point of view is an extremely significant component of advanced CASE tools.

First of all, GRAPES/4GL has several new types of diagrams when compared to GRAPES-86. The most important of them are screen forms (SF diagrams) and report forms (RF diagrams), serving as the graphic basis for advanced screen input/output and report generation facilities, respectively. The other area of the most impressive extensions is PD diagrams, where advanced facilities for form-based input/output and database management are added.

4. Screen Forms

First, we present a brief discussion of screen forms (SF diagrams). In order to make their use easier, the typical subcases of man-machine interaction - data input/output, fixed menu choice and selection from a data list are represented by separate subtypes of SF diagrams : I/O, MENU, SELECT respectively. Each screen form contains the active elements - fields (items) and the passive ones - texts, the fields being connected to program data elements during input/output. Important elements of forms are screen arrays, which are connected to data arrays, lists or sets thus permitting the representation of large data objects via easy predefined scrolling. The form sublanguage is closely related to data type definition facilities,, thus on the one hand, covering easily the traditional form generation in data base

management languages and, on the other hand, yielding much greater flexibility due to broader data type concept.

Fig.1. presents a typical example of GRAPES/4GL screen form.

However, these are not the screen forms themselves which give a significantly increased input/output functionality, but rather their close cooperation with I/O facilities in PD diagrams, which actually connect data to forms. These facilities will be described later in greater detail. But one important feature must be mentioned on the spot. Namely, the application-specific control of the data input or modification process is ensured by advanced set of input events generated by user actions. These events include AT START, AT FINISH, KEY keyname, BEFORE FIELD fieldname, AFTER FIELD fieldname, BEFORE ANY FIELD, AFTER ANY FIELD, BEFORE ROW, AFTER ROW, AT DELETE, AT INSERT.
The situation when an event occurs should be evident from the event name.

5. Report Forms

Report forms (RF diagrams) contain even more novel elements than screen forms and are more complicated in a sense. In addition to standard data layout specification they contain built-in facilities for row grouping and group break processing. Namely, group headers and footers in the form layout part together with the related grouping/ordering specification in the field description part completely specify the needed grouping in most standard cases. Computable form fields specified by expressions are used the same way as in screen forms. Thus the corresponding report generation statements in PD diagrams may be very simple, as a rule, just presenting the data (a set of records as a rule) on which the report is to be generated. Fig.2. presents a typical example of report form.

However, it is much more difficult to have an easy merge of standard report control mechanism with non-standard additional printing related to some data elements in the report. The solution is also presented for this problem in GRAPES/4GL. Namely, a report form also defines an event concept, a print interrupt in this case. The print interrupt is defined at the corresponding row in form layout (see the letter A in the example), and invokes the actual interrupt during report generation, when printing reaches this row. The application-specific interrupt processing is described in PD diagrams, in a very similar way to event processing during input/output. The regular printing defined by the form in this row may be freely combined with any desired non-standard printing in the corresponding event fragment.

6. Data Base Facilities

Though GRAPES-86 already contains the ER model concept, it is significantly extended on GRAPES/4GL, and plays a much more important role. The most significant extensions are nested entities and subtype entities. The graphic notation used for ER models is that introduced by J.Martin [13]. A new graphic element is the notation for nested entities by just placing them inside parent entities. Fig.3. presents a typical ER model in GRAPES/4GL - a standard sales department data base (with nested entities).

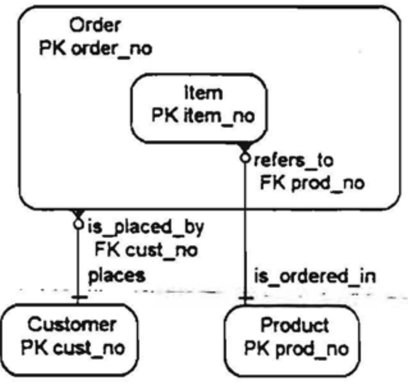


Fig .3

However, the main novelty is not in ER model internal features, but rather in its broad and consistent use, much deeper than one conceptual data model for a system. CD diagrams in GRAPES/4GL contain an explicit symbol for (passsive) data base, whose type is defined by the corresponding ER model. Access rights of each process to entities are described by special access paths and tables. Relationships are explicitly used in data manipulation, and their cardinalities serve as executable data base integrity constraints.

The traditional way of system implementation is to transform the conceptual ER model into relational models describing actual data bases. GRAPES/ 4GL approach is quite different, namely, to reference directly the components of ER model in data manipulation statements of PD diagrams (see some details later). Thus, a complete continuity from design phase to implementation phase in the data definition area is obtained.

7. Extensions to PD diagrams

GRAPES-86 PD diagrams are more or less traditional flowcharts from graphics point of view.

Grapes/4GL PD diagrams contain a number of new elements. From the graphic syntax point of view an important element is a statement body. It is used to

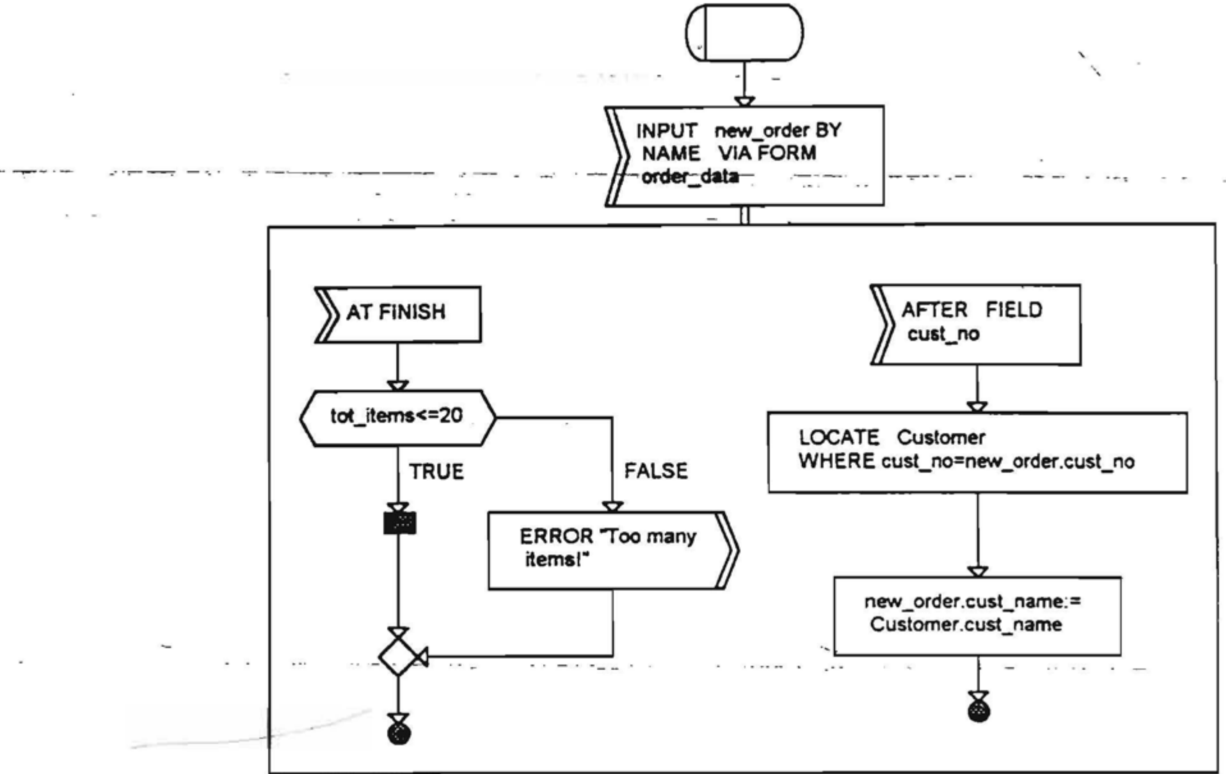


Fig. 4

group together repeated actions related to one statement. One use of statement body is a more readable graphic form for conventional WHILE and FOREACH loops. However, the main use of bodies is GRAPES/4GL screen input/output statements based on screen forms described above. Now let us describe the screen input statement in greater detail (the graphic syntax for other screen I/O and for report generation statements is similar).

Fig.4 presents an example of input statement for entering data via the form from Fig.1.

The statement consists of head and body. The head of INPUT statement performs the connection of fields from variable new_order (which is meant to be of an appropriate record type) to equally named form fields and it starts the statement execution. There should be an array or set field with name items in new_order, which is connected to the form array. The elements of the array must, in turn, be records with fields named like screen array fields; thus the connection of data elements to screen elements is performed up to the lowest level, using this by-name principle. For the simplest case with only default checks of entered data no body is needed. However, as a rule, problem-specified data checks are necessary. They are performed by means of event-related fragments in the body. Each event activates the fragment starting with the corresponding event symbol. For example, after the user has entered the field cust_no (i.e. he presses ENTER for that field), the event fragment AFTER FIELD cust_no is activated, which could, e.g. check whether such customer code is present in the data base (in the entity Customer). In addition the value of the form field cust_name may be supplied from the entity (by simple assignment to the corresponding field in the variable new_order). The fragment terminates either with CONTINUE (continue input) or EXIT (end the whole statement) symbols.

Each event fragment is independent of others, so it is more like a rule to be checked in the specified occasion.

The other group of statements, worthwile to be mentioned, is data manipulation statements. The functionality of these statements completely cowers SQL for the relational model. But their form seems to be more readable since it is based directly on ER-model. In addition, a based variable (in other words, automatically dereferenced pointer) is defined for each entity in a data base, and use of these variables seems to be easier than that of SQL cursors. For example,

LOCATE Customer WHERE cust_no = 125

immediately gives access to the desired instance of Customer entity (if there is such).

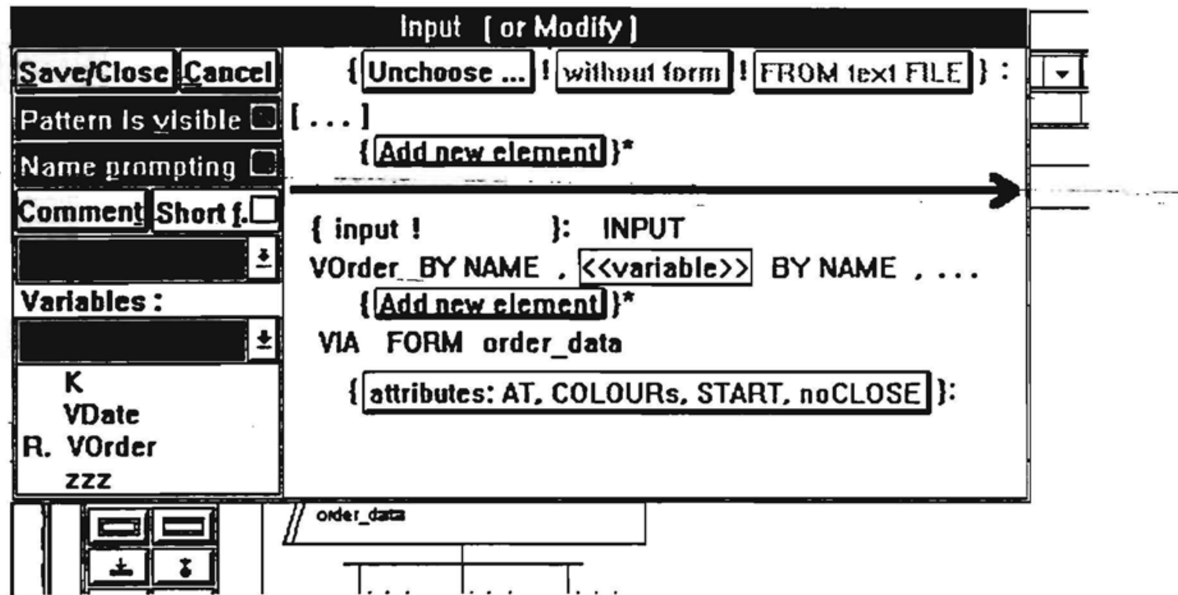


Fig. 5

Role name in relationships may be used for easier specifications of joins (using an extended syntax of SELECT expressions).

There is a lot of built-in functions and other details in GRAPES/4GL, but they are of a more technical nature to make the language a full-blown 4th generation data base programming language.

So to sum up, a language suitable for all development phases, from system analysis via design to implementation is proposed. Certainly, a subset of language facilities is used in each phase, but the actual bounds of each subset very much depend on development style and team individualities. For example, screen forms may be used to specify documents already in analysis phase (thus enabling early prototyping), or they may be introduced as actual forms during implementation.

Complete description of GRAPES/4GL language is given in [14].

8. Overview of GRADE WindowsToolset

The toolset contains the following components :

- graphic editors
- syntax analyzer
- language interpreter, supporting prototyping, simulation and debugging
- code generator for target environment, currently MS DOS or UNIX with INFORMIX as a standard SQL database.

All components are supported by common repository used for storing GRAPES models (diagram hierarchy together with diagrams themselves). A lot of common functions are supported by the repository, e.g. cross-referencing, supply of the lists of visible names etc.

Now let us characterize the most novel features of each component.

9. Graphic Editors

The main objective of graphic editors in GRADE is to support a graphic design and graphic programming in a manner as easy and simple for the user as the use of textual editors. To achieve this goal a lot of innovative features have been incorporated in the family of GRADE editors for all sorts of diagrams. Now let us characterize briefly the most important of them.

• Automatic allocation of elements.

Editors support automatic allocation of diagram elements, wherever it is reasonable. All this is done in a fast and effective manner. At the same time the user can combine the automatic allocation with a manual allocation for diagrams where some special outlook is of great importance. It should be noted that namely an efficient combination of both styles poses the most serious problems in editor design. Additional feature that the user can select manually is the diagram element style (colours, bold lines, shadows etc.)

Now let us be more specific for separate diagram types. The automatic layout is the main style for CD editor, however, the manual "prettydrawing" is also

supported for these diagrams when needed. The same is true for ER model diagrams. Process diagrams are allocated automatically, with the possibility to compress the layout afterwards.

The automatic allocation (for CD, ER editors) is based on special sophisticated graph theory algorithms (authors P.Kikusts and P.Ručevskis). These algorithms are very fast and yield an allocation which is both compact and easily perceivable from the user's point of view.

• Syntax promptin

Several prompting mechanisms are present from which the most remarkable is GRAPES/4GL PD statement syntax prompter. It should be noted that GRAPES/4GL PD statement syntax is rather complicated since it covers all features typically present in industrial 4GL's. At the same time it is very important for the user to start using the tools without a long learning period. A very convenient prompting is supplied for each statement allowing to enter the statement correctly as soon as the idea of the statement is grasped. The prompting is based on a set of predefined scenarios (generated from the language grammar).

More precisely, the prompting proceeds in the following way. First, all statements are grouped into large groups (according to the shape of their graphic symbol):

Decision
Case
Loop
Procedure
Receive
Send
Assign
Screen input
Screen output
Report
Data manipulation

The user selects the necessary group via icon or menu. Then the group is split into smaller ones using submenus (formed as dialog boxes). At last the user has selected the specific statement type he wants to enter. The prompter shows the general syntactic form of the statement.

The user then fills in the placeholders in the syntactic form or selects subcases of the statement by clicking on the appropriate selection marked by a representative keyword, e.g. [qualified] var BY NAME.

This immediately invokes the refinement of the template yet to be entered. A special provision is supplied for iterative parts of the statement ({add_new_element}*) where as many instances of that part may be generated as needed. Whenever the placeholder must be substituted by a defined name of the appropriate name class (variable, entity, type etc.), the visible names of that class are supplied. Thus even inexperienced users can enter a statement completely and without elementary syntax errors. Fig.5 shows an example of entering INPUT statement via syntax prompter.

The same principle is used to enter complex GRAPES/4GL expressions containing the numerous built-in functions.

The experience shows that prompting is principal for inexperienced users to have a quick start of GRADE usage.

10. Syntax Analyzer

The syntax analyzer is the most conventional component of all. It is implemented in a special compiler writing language RIGAL [15] (developed by M.Auguston and V.Engelson). Thus a high versatility (due to frequent changes in language syntax during development) and a high performance at the same time is obtained. The diagnostic messages are displayed to the user via the same graphic editors, in the case of PD diagrams using the same syntax prompter. Thus a precise error location and ease of correction is obtained.

11. GRAPES/4GL Language Interpreter

The language interpreter starts from the intermediate code prepared by syntax analyzer. This sole component is used for three related, yet different purposes: rapid prototyping of systems, system simulation for evaluating design solutions, and as a full-scale debugger for GRAPES/4GL programming. Due to this versatility a lot of execution modes has to be supported.

For prototyping and simulation, the process concurrency present in GRAPES has to be simulated. This is done in the most efficient way, completely preserving the GRAPES concurrency semantics at the same time (and allowing in addition for experienced users to have some control over the process scheduling).

All GRAPES/4GL screen input/output and report features are supported, thus an early prototyping of system user interface is also facilitated.

There are special provisions for system performance simulation. Though GRAPES/4GL is not a special simulation language, the same functional models with very little adds-on (like delay, workload etc. specification) may be used for performance simulation. Standard statistics are collected automatically in this mode and are stored as special predefined entities in data base. Thus special problem-related performance reports may be prepared easily by few GRAPES/4GL statements. The most commonly used statistics may be displayed in diagrams by graphic edtors.

When using GRAPES/4GL interpreter as normal language debugger, the main concerns are ease of use and complete semantic compatibility with the generated code. Both these requirements are strictly observed in the GRADE design.

12. Code Generator

Code generator starts with the same intermediate code as interpreter, and generates C code with embedded SQL statements. At present time the code is for MS DOS or UNIX environments, with screen input/output in text mode. INFORMIX data base engine is currently used for SQL support (with an easy switch to ORACLE possible). Code generator generates both data base definition/creation statements (from ER models of the data bases) and executable code for each independent execution unit (process in GRAPES terminology) of the system. The optimization level is sufficient for quite large systems to be completely generated tn this way. In case of need external C procedures may be linked in, e.g. to support interfaces to already existing parts of the system.

Complete description of GRADE tools is given in [16].

13. Conclusions and Future Work

The GRAPES/4GL language and GRADE tools described in the previous sections are currently being used for the development of several large information systems.

Currently the GRADE toolset is running in MS Windows. The minimum requirements for hardware platforms are 386-based machine with 4MB of RAM, with full MS Windows 3.1 support. However, for development of large projects 486-based machine with 8 or 16 MB of RAM is recommended. Then fairly good response time is achieved for large system models containing several hundreds of process/procedure diagrams and large ER models with several hundred entities.

Future perspectives of the GRADE development include Windows GUI style screen forms as the main extension of GRAPES/4GL language and support for team development of large models with the repository on network server.

Porting of GRADE to UNIX/MOTIF environment is also planned. We have also plans to incorporate OO ideas in the next versions of GRAPES/4GL and GRADE Tools.

14. References

1. J.Karlsson, A.Ek, " SSI - an SDL Simulation Tool", *Proc. 4-th SDL Forum*, North-Holland, 1989, pp.211-219.
2. V.Encontre, "GEODE : An Industrial Environment for Designing Real Time Systems in SDL" *Proc.4-th SDL Forum*, North-Holland, 1989,pp.105-117.
3. M.Dauphin, G.Fonade, R.Reed, "SPECS : Making Formal Techniques Usable",*IEEE Software*, November 1993, pp.55-57.
4. D.Harel, et.al. "STATEMATE: A working environment for the development of complex reactive systems, "*IEEE Transactions on Software Engineering*, 16(4), April 1990, pp. 403-414.
5. V.Berzins, Luqi, *Software Engineering with Abstractions*, Addison-Wesley, 1991.
6. A.Berztiss, *The Specification and prototyping language SF*, SYSLAB, Report 78, Royal Institute of Technology and Stockholm University, 1990
7. G.Martin, *Information Engineering Book* I-II-III, Prentice Hall, 1991.
8. P.Coad, E.Yourdon, *Object-Oriented Design*, Prentice Hall, 1991.
9. J.Rumbaugh, et.al.,, *Object-oriented Modeling and Design*, Prentice Hall, 1991.
10. G.Martin, *Principles of Object-Oriented Analysis and Design*, Prentice Hall, 1993.
11. G.Held (ed.), *Sprachbeschreibung GRAPES*, Siemens Nixdorf, 1990.
12. J.Barzdins, et.al., "Unified Specification Language and Integrated CASE Tools for Information System Development", *Proc. Baltic DB'94*, May 1994, Vilnius (to appear)
13. G.Martin, C. McClure, *Structured Techniques: The Basis for Case*, Prentice Hall, 1988.
14. *Grade V1.0 : Modeling and Development Environment for GRAPES-86 and GRAPES/4GL, Language Description*, Siemens Nixdorf, 1993.
15. M.Auguston, "Programming Language RIGAL "*ACM SIGPLAN Notices*, Vol.25, N12, December 1990, pp. 61-69.
16. *GRADE V1.0: Modeling and Development Environment for GRAPES-86 and GRAPES/4GL, User Quide*, Siemens Nixdorf, 1993.

Extensions of GRAPES/4GL for Windows style input/output

A.Kalniņš
Institute of Mathematics and Computer Science
Rainis boul.29, Riga LV-1459, Latvia

Abstract

The paper presents a short description of extensions made to specification and implementation language GRAPES/4GL in order to support Windows style graphic user interfaces. Both Windows style screen form definition facilities and new features in process diagrams supporting extended event processing and new control patterns typical to Windows interfaces are presented. Most of the GRAPES/4GL style proven valuable for existing textual I/O has been retained in the new version.

1. Introduction

A massive transition from textual interfaces to Windows style graphic user interfaces is going on for nearly all information systems in the recent years. Consequently, the same is to happen for tools supporting the information system development. The paper describes how this transition is being performed in GRAPES/4GL specification and implementation language and its supporting tool GRADE Windows [1].

One popular approach in this area is represented by system development languages based on SQL servers for Windows environment, like Microsoft ACCESS[3] or Gupta SQL [4]. The user interface sublanguages in these systems comply to Windows standards, but are completely SQL table based on the other hand. Simple table-related screen forms and their support logic can be generated very quickly this way, but it is not so easy to obtain more general and sophisticated graphical user interfaces by these means.

On the other extreme, there are object-oriented development interfaces above the standard MS Windows API, like Borland's Object Windows for Pascal [5] and Object Windows for C++[6]. Though these interfaces significantly simplify the application development when compared to standard API based tool kits, that approach, though universal, is still too complicated to be used on a broad scale for information system development.

The main objective during the design of graphical user interface facilities in the new version of GRAPES/4GL (called GRAPES/4GL-W in the paper) was to find a compromise between the two above mentioned extremes, or in other words, to define a language simple enough for standard cases and flexible enough for sophisticated user interface definitions really appearing in information systems. The major goal was also to maintain the continuity with existing GRAPES/4GL text based user interfaces, to facilitate a semiautomatic conversion of these interfaces to graphic style.

The paper describes the graphical screen form definition sublanguage based both on CUA standards [7] and UNIX/Motif environment Style guides [8]. The available attributes of forms and their elements are specially tailored to the needs of information systems. Properties required to define, e.g., or full scale graphic editor are intentionally omitted as too complex.

Then the GRAPES/4GL-W statements to be used in process diagrams (PD) for controlling the screen forms and relations of these forms to data (the widely used GRAPES/4GL connection principle) are described, as well as broad facilities for describing reactions to events generated by users interacting with the forms on the screen. Namely, the event processing is the most essential part of Windows programming, and it is where a lot of innovative features have been incorporated in GRAPES/4GL-W . A short description of more technical features like statements and built-in procedures for updating the form and element attributes is presented, but the role of these functions is significantly less significant here than in, e.g., Object Windows [5].

2. Graphic Screen Forms

Though graphical user interface elements are strongly determined by documents such as CUA definitions [7] and Motif style guides [8], incompatibilities in these documents leave some room for choice. In GRAPES/4GL-W namely those elements have been chosen, which are supported by all Windows style environments, and which are of some significance for information systems. New elements in

GRAPES/4GL-W are combinations of the standard ones and represent a typical building blocks of user interfaces.

It should be mentioned that some influence in the selection of elements has come from the product Dialog Builder being developed by Siemens Nixdorf [9]. That approach also has a goal to unify different Windows environments, though it has no sufficient orientation towards information systems.

2.1 Screen Form Types in GRAPES/4GL-W

Like in GRAPES/4GL , screen forms are defined by diagrams of type SF, however with completely new subtypes.

The main division of forms in GRAPES/4GL-W is into ***non-modal*** and ***modal*** ones . The non-modal forms are normally called ***windows*** and have the property that one application may have several windows open simultaneously with the possibility for the application user to select which window to interact with. Modal forms are often called dialog boxes also, and only the current one of them may be accessed by the user. Non-modal forms are further divided into the following subtypes:

- ***main windows*** (with subype name MAINWIN). There may be only one main window per application, it is open for the application's lifetime and controls all other forms of this application, its closure terminates the application.
- ***start windows*** (with subtype name STARTWIN). Start window is displayed at the beginning of application execution (visually before the main window) in order to present e.g., the company's logo or some animation. There are special closure agreements for this type.
- ***windows*** (with subtype name WINDOW) are ordinary windows for exchanging the information with the user, there may be many windows open in the same application, including instances of the same form. The user may manually select the window for interaction.
- ***subwindows*** (with subtype name SUBWIN). They are used to structurize complex window definitions with large number of elements. Each subwindow is described by independent SF diagram, but actually it is a fixed part of parent window, represented by subwindow reference in it.

Modal forms (with subtype name DIALOG) in GRAPES/4GL-W remind most closely the previous I/O forms in GRAPES/4GL. The user can communicate with only one modal form at a time, and that must be closed before another one may be accessed. Modal forms are normally used for input or modification of limited amount of data. It should be noted that other GRAPES/4GL form subtypes (SELECT, MENU) have become just elements of GRAPES/4GL-W forms.

2.2. Form Elements

GRAPES/4GL-W forms consist of elements. Only three kinds of them: fields, texts and field arrays are actually inherited from previous GRAPES/4GL form elements, the other ones are implied by Windows GUI standards. Elements are of two types: data elements used for data input or output and command elements used to generate events related to user activities. Data elements, in turn, are active ones, which may be used both for entering and displaying data, and passive ones, which may be only displayed.

The active data elements are:

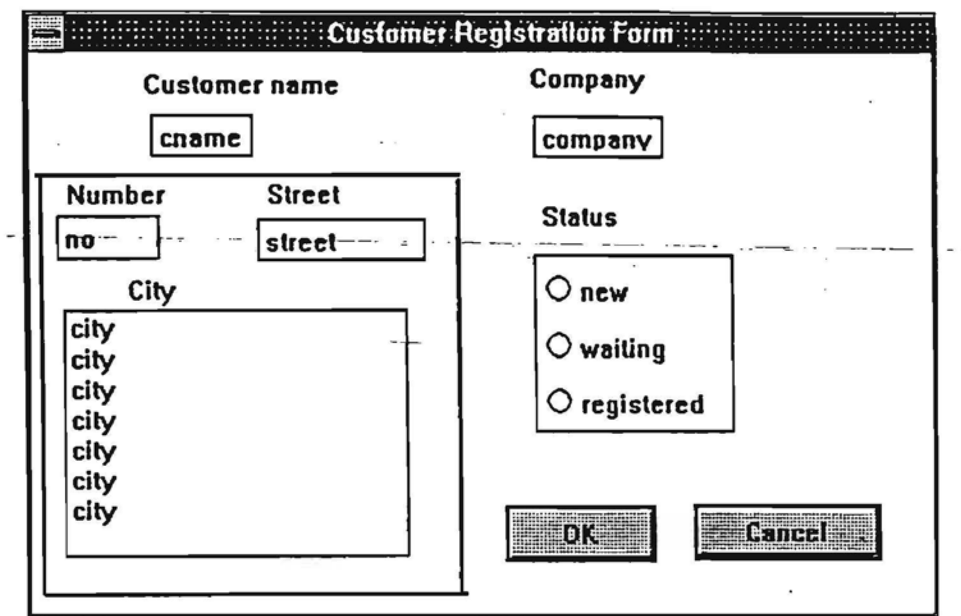
- *field* (for elementary data I/O of any type,
- *listbox* - for selection from a list,
- *combobox* - for data selection or entry,
- *scroller* - for selection of a numeric value within a range,
- *radio button group* - for selection of a value from a set,
- *checkbox group* - for setting a group of flags,
- *field array* - for I/O of an array of records.

The passive data elements are:

- *text* - for display of texts,
- *bitmap* - for display of graphic bitmaps,
- *drawing area* - for display of graphic information via special drawing functions,
- *status line* - for displaying status messages,
- *frame* - visual element (line or rectangle) for framing other elements.

The only element not traditionally proposed by Windows standards is field array. It has the same structure of equal rows of fields as in GRAPES/4GL, only a scroller is added for moving the visible part of a larger data array, list or set. All active data elements are meant for connecting them to variables or their parts at I/O statements, thus the proper data I/O occurs via these elements. Listbox and combobox elements actually have two subparts: the ITEMS subpart (referenced via special syntax listbox_name:ITEMS) for displaying a selection list and the selection subpart named as the element itself for returning the selected value(s).

Passive data elements are used normally as fixed decorative elements whose values are set by special functions. For example, texts may be changed, one of several bitmaps made visible, and so on.




```
DIALOG cust_reg,
    POSITION=(65,72, 473, 294),
    TITLE="Customer Registration Form",
    AUTORESHOW,
    TYPE REFERENCE = cust_reg_t  BY STRUCTURE;
    ELEMENT LIST
        FIELD cname, POSITION=(63,33), REQUIRED;
        FIELD company POSITION=(362,33), REQUIRED;
        TEXT T1, POSITION=(48,9), VALUE="Customer name";
        TEXT t2,POSITION=(255,9), VALUE="Company";
        GROUP caddr
            FIELD no, POSITION=(14,86,53);
            FIELD street, POSITION=(118,86);
            LISTBOX city, POSITION=(18,136,177,135);
            TEXT t3, POSITION=(14,67), VALUE="Number";
            TEXT t4, POSITION=(118,67), VALUE="Street";
            TEXT t5, POSITION=(21,117), VALUE="City";
            FRAME f1, POSITION=(4,64,218,220);
        END GROUP caddr;
        RADIOBUTTONS status, POSITION=(262,107),
            ORIENTATION=VERTICAL,
        RBUTTON LIST :
            RBUTTON b1, VALUE=new, TITLE="new";
            RBUTTON b2, VALUE=wait, TITLE="waiting";
            RBUTTON b3, VALUE=reg, TITLE="registred";
        END RBUTTON LIST;
        BUTTON OK, POSITION=(246,238), TITLE="OK", DEFAULT;
        BUTTON Cancel, POSITION=(346,238), TITLE="Cancel";
    END ELEMENT LIST;
```

Fig.1

The command elements of GRAPES/4GL-W are

- *button* - standard Windows button,
- *pull-down menu* - a menu element of the window positioned according to Windows standards,
- *popup menu* - a command list which pops up when right mouse button is pressed,
- *accelerator* - a key combination used as a shortcut for other command element.

It should be noted, that according to Windows standards, GRAPES/4GL menu forms are to be transformed in GRAPES/4GL-W either to pull-down menu elements of some window, or to pop-up menus appearing on user's demand.

A special element of a window is reference to a subwindow which is described in a separate screen form.

Yet another special element is *group* which has no graphic appearance, but only defines a list of elements (or other groups) contained in it. Group is introduced to facilitate structured data connection to SF elements and common actions with several elements. Group concept is an analogue to record concept in data type definition.

Each form element has a name, which must be unique in the form (or comprising group, if there are groups defined). Elements are referenced by their names, which may be qualified (by group names).

2.3. Form and Element Attributes

The form as a whole and each of the elements has attributes which specify both the graphical and logical properties of it. The graphical attributes like *x-, y-coordinates, height, width, colour, border* etc. actually are implied by Windows standards. A number of logical attributes are induced by GRAPES style of interaction with elements and they are common to all elements, like: *visible, enabled, autoreshow, required, helpid*. These attributes determine the rules of entering the element value by user or the rules of displaying the value. Each type of element has also its specific attributes. For example, *field*, in addition to above mentioned, has attributes *default, expression, font, word-wrap, alignment, password, multiline, maxlen, type*. These attributes partially are inherited from GRAPES/4GL and partially imposed by Windows standards (on Edit control, which is the same).

Each attribute has a specific value range, mainly integer or yes/no, however for some attributes larger fixed value sets or string type values apply. Attribute values for the form and its elements are mainly defined at creating the form diagram via the appropriate GRADE editor. However, some logical and even graphical attributes may be changed by special statements and functions during execution, for example, field may be moved or resized, it may be made visible/invisible, enabled/disabled, required/optional.

The main objective in defining the attribute set in GRAPES/4GL-W has been to make as much properties of elements statically definable at form design as possible, in order to simplify the corresponding PD diagrams.

As in the previous version of GRAPES/4GL, SF diagrams contain the graphic layout part and textual description part where attributes of the form and all its elements are visible. If an attribute is not present in the description, it has the default value. The four geometrical attributes: x,y, width and height are gouped together as position attribute in the textual form. The presence of a yes/no type attribute means the "yes" value for it, with "no" values coded by the corresponding "opposite" keywords. Fig.1 shows an example of GRAPES/4GL-W screen form - a modal form, as it will be entered by next version of GRADE editors.

3. Input/Output Statements in GRAPES/4GL-W.

The main concern in I/O statement design has also been the balance between the simplicity and vast possibilities of form behaviour typically to be found in Windows environment. The main idea was to preserve general mechanisms in GRAPES/4GL - the connection of data to forms by major input/output statements and processing of input/output details in event fragments, in response to events generated by user actions.

3.1. Main Input/Output Statements

Modal forms in GRAPES/4GL-W have in general the same interaction logics as I/O forms in GRAPES/4GL. Therefore the same *INPUT, MODIFY* and *DISPLAY* statements are retained in GRAPES-4GL-W for modal forms. The statements open the form and connect data to form elements. When the user has ended the data modification or entry, and has pressed the OK button, the statement is also ended. Fig.2 presents the simplest example of how the new value of customer_record variable (*cust_rec*) is entered via the screen form in fig.1. Here *cust_rec* is assumed to have record type with fields *cname, company, caddr, status*, with *caddr* being a record in turn with fields *no, street, city*.

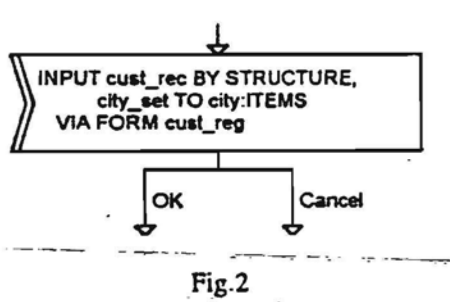


Fig.2

The statement looks much like as in previous GRAPES/4GL version, except that new connection type BY STRUCTURE is used instead of BY NAME (due to the fact that the form has a group) and the special syntax for setting the value of ITEMS list of the *city* list box. The details of the extended connection concept in GRAPES/4GL-W are presented in the next subsection. Two exits of the statement correspond to two possible buttons for ending the input.

However , it is not very typical in Windows I/O style to have no event processing for an input. Event processing may be done either in statement body, as in GRAPES/4GL, or in the special event procedure (which actually is a remote body, with some extended features). The event processing mechanisms are described in detail in 3.3.

Thus, the complete syntax of statements for modal screen forms is:

{INPUT | MODIFY | DISPLAY}
connection_list
VIA modal_screen_form_name
[WITH event_procedure [(param_list)]]

The last option is used to reference the event procedure (the remote body), which may even have its explicit parameters (e.g., when common event procedure is used for several statements relying on the same screen form).

The statement names express the same traditional semantics from GRAPES/4GL, however, the differences are no more so significant (input may be more freely mixed with output in Windows style).

Non modal forms, i.e., windows, have a completely new interaction logic for GRAPES/4GL. They remain opened for a long period of time, and, what is more essential, there may be several windows for an application opened simultaneously, with the possibility for the user to select the window he wants to interact with. Thus a new element of quasiparallelism is introduced, since the bodies (or event procedures) for all the open windows must be ready to work. When several windows of the same type are open (e.g., to process several related documents of the same type), there must be instances of the event procedure, each having its own data. Due to all this, a new statement *SHOW* is introduced for windows.

Due to several subtypes of windows, there are several typical cases of SHOW usage. First, it is very typical in Windows application, that the corresponding process has the simplest structure depicted in fig.3, containing only SHOW for the applications main window - the form *mainwind*.

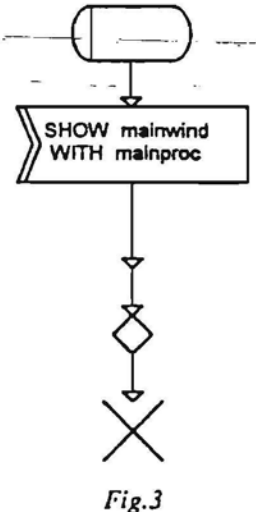


Fig.3

All the real job done by the application is performed by the event procedure *mainproc*, which responds to events invoked by user activating the control elements of the main window - selecting menu items, pressing buttons etc. Each of the responses certainly may be as complicated as necessary and invoke any other I/O. The structure of the process may obviously be also more complicated and some preparatory and conclusive data-related activities may be present before/after the SHOW statement. However, any I/O related activity must be within the event procedure. It should be noted, that a typical Motif application has no main program at all, it is just a "pile of call-back-routines", a style, which seems not very appropriate for GRAPES/4GL.

If the start window is to be shown before the main one, the corresponding SHOW statement is to be placed in AT START event of the main window.

The most complicated form of SHOW statement is used for (simple) windows, several of which may be open. The full form of SHOW statement is:

SHOW [connection_list IN]
non_modal _screen_form_name
[WINID winid_var]
[WITH event_procedure_name [(param_list)]]
[FOR duration_expr]
[NOCONTINUE]

The connection_list is the same as for modal forms, but it is used only if the window actually has data elements. Window identifier is used if there is a need to reference one of several open instances of the same window form (otherwise the form name is sufficient for reference to

the window). The usage of event procedures is the same as for INPUT, but it is even more typical here due to typically larger event processing (however, event processing in body also may be used). FOR-expression may be used for windows to be visible for a limited time. The NO CONTINUE option (which is by default for main window) tells the statement to wait until the window is closed and only then proceed to the next statement. However, without this option the window is opened, event procedure (or body) instance is activated and just after this the next statement is started, thus allowing, e.g., the same SHOW statement to be executed once more after a while.

3.2. Connection Principle

As it was already mentioned, the connection principle of GRAPES/4GL is taken over and extended to cover the new form features also. The principle lies in the following that an appropriate variable is connected to each element of the form by connection list of the input/output statement. This connection, in general, lasts while the statement is executed (or window is open for SHOW statement). Very typical case is that one record variable is connected to the whole form, with record components connected to the corresponding elements, as it was demonstrated in fig.2. To explain the connection more closely, the type structure of variable *cust_rec* (used in fig.2) is presented in fig.4.

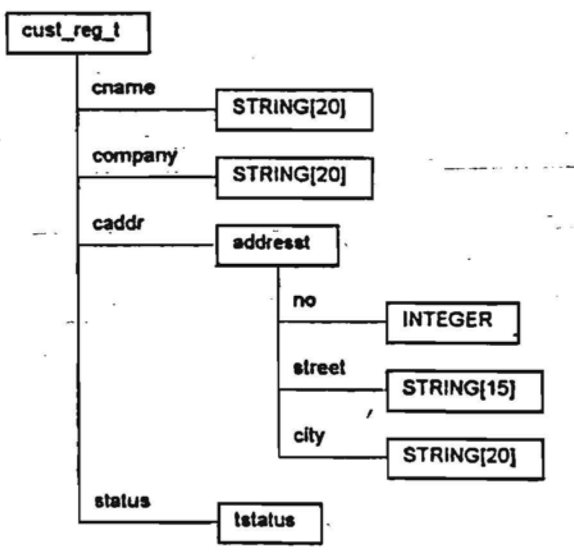


Fig.4

If we consider once more the screen form, with the group *caddr* in it, it is easily observable that the structures of the form and record type are isomophic, with elementary record fields corresponding to form elements, and subrecord to group. Just to represent this very typical situation, a new connection type *BY STRUCTURE* is used in the example in fig.2. Certainly, the *BY NAME* option as in GRAPES/4GL may also be used. The difference is, that BY NAME completely ignores subrecords on the one side, and groups on the other, while BY STRUCTURE takes them into account. In the very simple example of fig.2, *cust_rec* BY NAME could be used as well, since all elements are unique formwide here, but it is not always the case.

Variables and their parts may be independently connected to separate elements also. In that case, the same BY NAME, TO and FROM options may be used as in GRAPES/4GL. However. element names may be qualified by group names, e.g., *city1* FROM *caddr.city*. For BY NAME option, additional {TO|FROM} group_name clause may be used, to indicate the desired group if the element name repeats in the form, for example, *no1* FROM *caddr* BY NAME. It should be noted that group names also may be qualified (in case of nested groups), and {TO|FROM} group_name clause may be added for record connection also, before the BY NAME or BY STRUCTURE clause.

Now some words about the elementary connection to form elements. Any elementary data type may be connected to field, combobox, single selection single column list box. Numeric types may be connected to scroller, and enumerated types or types with defined VALUE_SET attribute to radio button group.

List, array or set (of records) may be connected to field array. The same is true for listbox: ITEMS of a multicolumn list box, while list, array or set of elementary-typed elements must be connected to combobox: ITEMS. List or set may be connected to multiselection listbox. Apparent integrity rules apply for objects connected to listbox and listbox:ITEMS respectively (the same is true for combobox). It should be stressed that list box: ITEMS normally is connected by TO option, even in INPUT statements, since that connection always is of display type. We conclude with the remark, that a special CONNECT statement is introduced in GRAPES/4GL-W to change connection, e.g., inside SHOW statement.

3.3 Events and their processing.

As it was mentioned in 3.1, Windows style input/output is highly event based, and GRAPES/4GL-W provide a lot of new features for event processing. The event concept per se is the same as in GRAPES/4GL, only the event list is significantly uppgraded and new event management mechanisms introduced.

The statement body as a container for input event processing remains valid also for GRAPES/4GL-W. However, since the body could become enormous

and contain the whole program (see example of fig.3), a new concept of event procedure is introduced. It is a PD diagram of new subtype EVENTPROC. Its syntax is similar to existing MACRO subtype, in the sense it contains disjoint event fragments. However, the diagram of type EVENTPROC is also similar to procedure, in that it may have its local DT and SD (formally it has SD always, but there may be also nontrivial SD with parameters).

An event procedure is referenced in WITH option of the corresponding I/O statement, in that case the statement may have no body, since all event processing is then done in the event procedure. Event procedure may also be activated by CALL statement in a body or other event procedure, then the relevant event processing is delegated to this subordinated event procedure (some restrictions apply to this case, see later).

Variable visibility within event procedures is the standard one for GRAPES/4GL. Event procedure's DT describes its local data, with lifetime of the statement execution (for modal forms) or of the corresponding window lifetime (for non-modal forms). Each event procedure instance for an open window has also its own instance of the local data (the subordinated event procedures behave correspondingly). Certainly, if some process data are connected to a window in SHOW statement, no instances of that data are generated. To save these data in the local instance, event procedures with formal parameters are to be used. Parameters also have the lifetime of the instance (i.e., they are copies of actual parameters). Actual parameters may be supplied explicitly in WITH option, but an additional arrangement is provided also. If parameters are specified in SD, but not in WITH option, data elements from the connection list are taken as actual parameters (in their natural order, disregarding BY NAME, BY STRUCTURE, TO, FROM options).

Another (and a very traditional for Windows) possibility is to obtain necessary data for a window instance during start event processing, store them in local variables of the event procedure, and connect them to visible elements using CONNECT statement(see 3.4).

The event fragments in general look like those in GRAPES/4GL. They start with event symbol specifying the event, e.g., AT START, AFTER *cname*, may contain any GRAPES/4GL-W activities including nested I/O, and end with either CONTINUE or EXIT symbols. CONTINUE just ends the fragment, while EXIT ends the whole statement (or closes the window, respectively).

The event types roughly correspond to Windows messages (Motif callbacks), however, only those of sufficiently high level and relevant to information systems are included. Thus, actually the type list is not so different from GRAPES/4GL.

The following events are defined for any form (modal or non-modal):

AT START
AT FINISH

AT START occurs after the form is opened, AT FINISH when it is to be closed. If AT FINISH ends in CONTINUE, closing is postponed (an equivalent to Windows CanClose returning false).

For windows, two events related to obtaining focus/loosing focus are defined:

BEFORE FOCUS
AFTER FOCUS

A group of events is related to form elements. All active data elements define events:

BEFORE element_name
AFTER element_name
AT CHANGE element_name
DOUBLE-CLICK element_name

Before- and after-events are related to input focus moves, AT CHANGE is invoked by value change of any type. For elements with keyboard input, an event

AT INVALID DATA element_name

is defined for programming application-specific corrective actions. For field array there are two specific events:

AT INSERT element_name
AT DELETE element_name

For command element, only the event

ACTIVATED element_name

is defined. A qualified element_name may be used, e.g., for multilevel menu items.

There is a special event for any element (or group)

EVENT FOR element_name

used only for structuring the event processing. Only a call to subordinated event procedure may follow this event (and this is the only place where such a call may be). Then any event processing for the given element is delegated further.

Some auxiliary events may be used to coordinate and control several open windows. They are:

MESSAGE message_name [INTO variable]
TIMER timer_name
AT duration_expr IDLE

Messages are sent and timers are set by special new modifications of GRAPES SEND statement (for sending inter-window messages and setting local (clock-based) timers).

3.4. Special Statements and Functions

From the vast number of functions and procedures, used in Windows to control forms and elements, not so many are actually needed in GRAPES/4GL-W. This section presents only a brief

overwiew of special statements and functions in GRAPES/4GL-W.

NEXT statement is used for forced move of input focus to form or element. The statement MAKE ELEMENT element_name attribute_list is used to change element attributes of type yes/no e.g., to make element visible/hidden, enabled/disabled etc. Attributes of a group of elements may also be changed.

MOVE and RESIZE modify the geometric attributes of forms and elements.

The RESHOW statement is used for forced immediate reshow (i.e., of updating the visible image according to the connected value) of an element or form. It is used when automatic reshow is not sufficient.

The statement CONNECT connection_list is used for changing or setting the element connection inside body or event procedure, e.g., for changing the selection list in listbox: ITEMS.

Special functions and procedures are mainly used for changing the (partially entered) element value while input focus is still on the element, or obtaining some specific attribute values. Some typical samples are functions:

GET_VALUE (field_name): string
GET_CURR_POS (field_name): integer
GET_SEL_POS(field_name) : integer
IS_VALID (field_name) :boolean
and procedures
SET_VALUE (field_name, string)
SHOW_STATUS (string)
etc.

The complete list, though small when compared to Windows, cannot be presented here for the sake of place.

4. Conclusions

Extensions of GRAPES/4GL language for supporting the Windows style graphic user interface have been briefly described in the paper. They require no significant changes in the kernel of language. The new extended version of GRAPES/4GL (GRAPES/4GL-W) is to be supported by GRADE Windows toolset [1] by the end of the year. The most significant effort will be necessary for the design of new screen form editor, where a lot of services are to be built in, e.g., for automated element /group structuring basing on a record type definition. The I/O part of the prototyper/debugger is also to be redesigned, and code generation will certainly support the MS Windows target environment.

5. Acknowledgements

I would like to thank Viktors Supe for numerous discussions and suggestions during this work.

6. References

1. J..Barzdins, A.Kalnins et. al. GRADE Windows : an Integrated CASE Tool for Information System Development, this volume.
2. *GRADE V1.0 : Modelling and Development Environment for GRAPES-86 and GRAPES/4GL, User Quide*, Siemens Nixdorf, 1993.
3. *Microsoft Access Step by Step*, Microsoft Press. 1993
4. *SQL Talk\Windows. Users Guide*. Gupta Technologies Inc, 1990.
5. *Object Windows. Programming Guide*. Borland International Inc., 1992.
6. *Object Windows for C++. Users Guide*. Borland International Inc., 1991.
7. *Systems Applications Architecture Common User Access. Advanced Interface Design Guide*. IBM, 1989.
8. *OSF/MOTIF, Style Guide*. Open Software Foundation, Prentice-Hall, 1991.
9. *Dialog Builder/Windows V2.1. Benutzerhandbuch*. Siemens Nixdorf, 1993.

PROCEEDINGS Janis Barzdins

The 7th International Conference on Software Engineering and Knowledge Engineering

Sponsored by

Knowledge Systems Institute (Founder and Organizer)
The Johns Hopkins University Applied Physics Laboratory
University of Pittsburgh

In Cooperation with

ACM (SIGSOFT)
IEEE Computer Society (TC on Software)

Technical Program, June 22-24, 1995

Rockville, Maryland, USA

RULE-BASED APPROACH TO BUSINESS MODELING

Janis Barzdins, Guntis Barzdins, Audris Kalnins

Institute of Mathematics and Computer Science, University of Latvia
Rainis blvd. 29, Riga LV-1459, LATVIA
E-mail: guntis@mii.lu.lv

Abstract

A system description model, sometimes called "business model" is considered. The business model describes the separation of the whole system into individual tasks and the sequence of these tasks. A completely formal rule-based language for precise description of tasks in the business model context is proposed. The language includes means both for precise description of task triggering conditions and of actions performed by the task. Finally the semantics problems caused by concurrent functioning of rules are discussed.

1 INTRODUCTION

By *business model* we understand a model unifying three widely accepted paradigms: dynamic model, data flow diagram and sequence chart .It is becoming widely accepted that the design of complicated information systems has to start with the business modeling of the system [1,2,3,4]. However, the concept of the business model itself has not completely established yet. Different authors put slightly different meaning into this concept. But all these approaches have one thing in common, namely, they are semiformal. A natural question arises how to formalize such a model completely.

Roughly speaking, the business model describes the separation of the whole system into individual tasks and the possible execution sequences of these tasks. The basic problem which arises there is how to formalize the individual task in the context of business model. A complete formalization of tasks in the form of rules is proposed.

The rule-based approach used here is influenced by a number of papers where the rule-based system description principles are elaborated. Rule-based approach used here is especially influenced by [5] where a rule-based language is applied to the specification of external behavior of the telephone exchange. Rule-based approach applied to information systems is studied in [6,7,8,9,10]. In terms of [7,8,9], the rule-based approach used in this paper may be considered as a further development of the "action rules".

2 BUSINESS MODEL: STRUCTURAL DESCRIPTION

We will start the definition of the business model (like in [4]) with the definition of Task Communication diagram which describes the separation of a system into individual tasks and the interfaces between these tasks. Formally we define the Task Communication Diagram as a graph containing four types of nodes:

- tasks,
- message queues,
- data stores,
- environment

and two types of edges :

- message routes
- access paths.

Figure 1 shows an example of a Task communication diagram, where T1, T2, T3 represent tasks, M1, M2, M3, M4, M5 - message routes and E1 - data store.

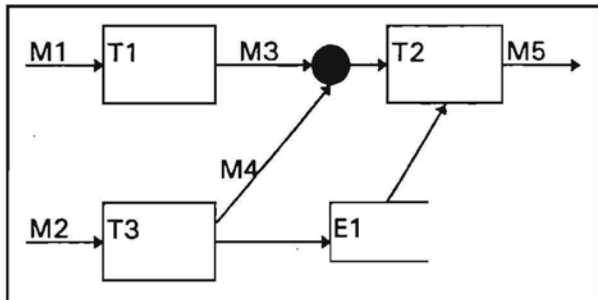


Figure 1. Example of Task Communication Diagram

We would like to stress one important difference between the Business model and the classic Data flow diagram. If in a Data flow diagram messages only denoted data transfer between the tasks, then in a business model messages also pass the control between the tasks and may trigger the tasks if their triggering conditions are fulfilled.

3 TASK: BASIC PROPERTIES

The main problem is to formalize the tasks. The nontriviality of this problem is caused by the following:

- tasks have to be activated automatically, thus task triggering conditions must be formalized,
- tasks have to perform actions which may continue for an unlimited time period; thus task instances are necessary,
- task instances have to perform independently from each other and concurrently.

The basic idea of our approach is that we define tasks in the form of special rules. For the sake of simplicity we assume that one task corresponds to one rule (though in general case one task may consist of several rules).

4 RULE: GENERAL SYNTAX

The general syntax of a rule is shown in Figure 2.

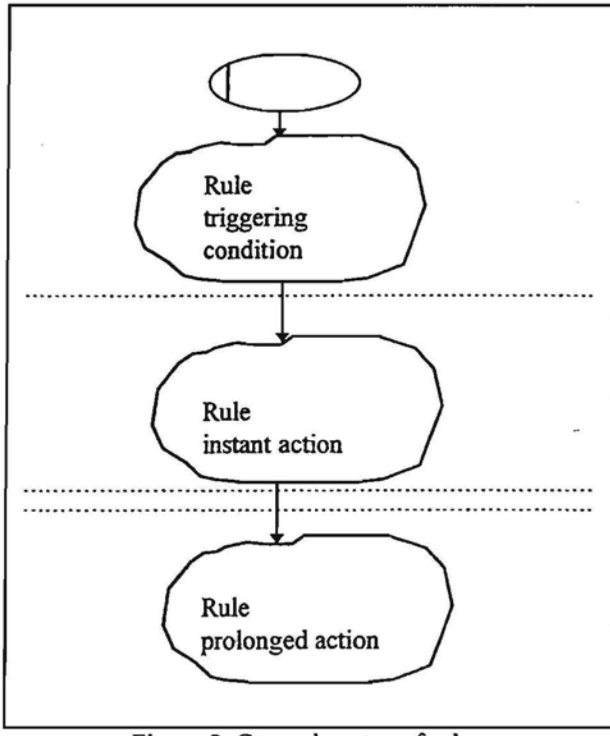


Figure 2. General syntax of rule

Rule contains :

- triggering condition
- rule instant action
- rule prolonged action.

Rule triggering condition is a logical expression in the graphical form built from the events. If this expression becomes *true*, the rule action is started.

Rule instant action is the action which executes instantly, and it is executed in the same time moment *t* when the rule triggering condition is checked to be true. Instant action may not contain WAIT statements.

Rule prolonged action is the process which is started after the rule triggering condition is fulfilled and rule instant action is completed. Rule prolonged action describes an activity which can execute for an unlimited time period. In other words, rule prolonged action may contain WAIT statements. By WAIT statements we understand the waiting statements for various events.

Motivation for the prolonged action will be given later.

5 RULE TRIGGERING CONDITION

We propose four types of events for describing triggering conditions:

- message events
- entity events
- time events
- data events (called also data conditions)

Rule triggering condition is defined as a Boolean expression in graphical form built from these types of events. Figure 3 shows an example of rule triggering condition. It corresponds to conjunction:

(*.*.15) AND (((Order O AND (Customer C AND C.Status=1) AND Product P) AND O.Name=C.Name AND O.Id=P.Id)

In the natural language this condition would be phrased like that:

If current date is *15*, and if *O* is some instance of message *Order* currently in the input queue, and *C* is some instance of entity *Customer* such that its field *Status=1*, and *P* is some instance of entity *Product*, and *O.Name=C.Name*, and *O.Id=P.Id*, then for such triplet *<O,C,P>* the rule action has to be executed.

Such "translation" of the triggering condition into natural language already provides some idea about the triggering condition semantics.

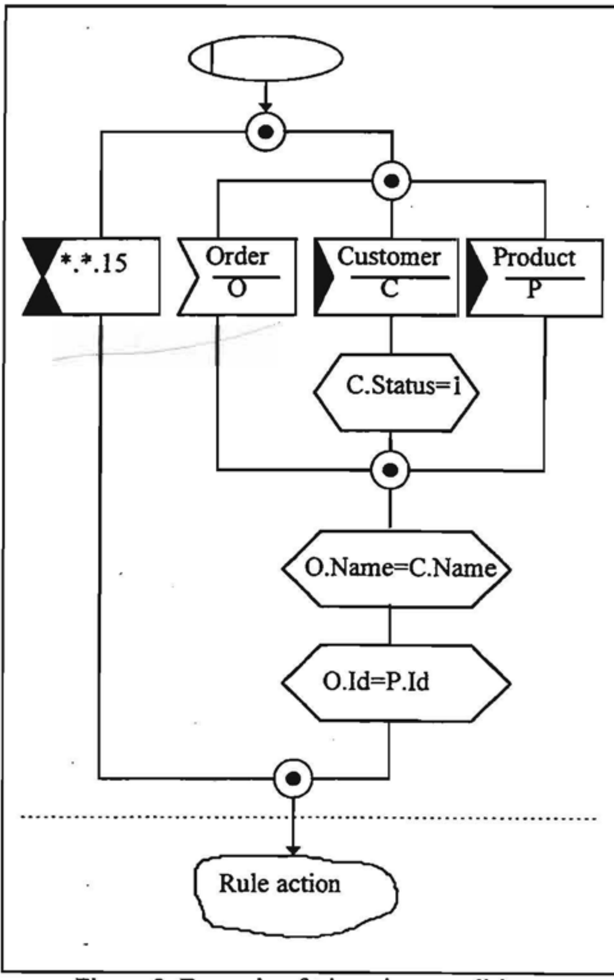


Figure 3. Example of triggering condition

Advanced facilities for describing events and triggering conditions composed of them are proposed. For example, a special sublanguage is proposed for describing time events. Quantifiers for expressing global conditions on data stores are also introduced. In complicated cases the semantics of triggering conditions requires a detailed explanation which is omitted here.

6 RULE ACTION

For the description of the rule action we have to choose some primary language. In our case as a primary language we will use GRAPES/4GL (see [12]) which is a well developed graphic specification and programming language. For those who are not familiar with GRAPES/4GL, here we will mention the main GRAPES/4GL statements:

- WAIT statement - for waiting for a message to appear.
- SEND statement - for sending a message.
- Besides the above mentioned statements, in GRAPES/4GL there are available also traditional data manipulation statements (assignment, case, loop etc.) and statements for interaction with Entity Relationship database.

Now we can describe in more detail the action part of the rule. As it is shown in Figure 2, there are two types of actions triggered by the rule condition: instant action and prolonged action

Syntactically the instant action is any program fragment in the underlying language (in our case it is GRAPES/4GL) which contains no WAIT statements. Prolonged action may contain WAIT statements. Roughly speaking, the prolonged action is the same GRAPES/4GL process.

Now some words about the motivation why instant and prolonged actions have to be separated:

As it was stated before, the instant action executes instantly in the same moment when the Rule triggering condition is checked to be *true*. On the other hand, the prolonged action may function for unlimited time period. Such division of the rule action in two parts has the following reason. Frequently, immediately after checking the rule triggering condition being *true*, we may want to perform simple data manipulations before other rules functioning concurrently have "corrupted" the data - such statements have to be included in the instant action which by definition is executed at the same time moment as the triggering condition is checked. On the other hand, it is clear that we cannot get around with the instant action alone, because inside the action part we may want to describe also message waiting and other prolonged actions. For example, if we want to describe a telephone conversation by a rule, then it is evident that waiting for messages in the rule body is absolutely necessary. A special property of prolonged action is that several instances of the same prolonged action may execute in parallel, for example, servicing separate telephone calls.

7 SEMANTICS

From the semantics point of view, the Business model is a set of rules:

$$\{R_1,\ldots,R_n\}$$

In the previous sections, where we defined Rule syntax, we also partially defined their semantics. This semantics we will call "naive" semantics. In most cases it is sufficient for describing of the system by means of rules. But there are cases when this "naive" semantics is not sufficient. Therefore a more precise definition of the semantics is necessary.

But before we pass to the main principles of this definition we will point out some problems making this definition nontrivial.

The main problems are caused by the prolonged action, which may contain unlimited waiting. Therefore several rules may perform concurrently. Moreover, whenever the triggering condition of a rule is true, a new instance of the rule is activated whose prolonged action again may perform for an unlimited time. Thus, according to our "naive" semantics, several instances of the rule may be created and perform concurrently. The main problem here is to define the instance creation mechanism in a precise and consistent way.

Another nontrivial semantics problem is related to a time event, i.e., exactly when and how long this event is true and invokes the triggering of new rule instances. A similar problem is related to data conditions due to several entity instances. The main difficulty here is to define the exact triggering semantics so that the creation of unreasonably many rule instances may be avoided.

Exact and consistent solutions to all these semantic problems are proposed in the semantics definition of the rule language. This definition is an operational one, i.e., an interpretative concurrent model for language execution is proposed. This model corresponds to the mentioned "naive" semantics in all normal cases.

8 CONCLUSION

The paper outlines the ideas of a rule based language for business modeling more from the theoretical point of view. However, pragmatically the goal of this study is to incorporate Business modeling and Rule-based approach in the family of GRAPES languages [11,12] and in the next generation of GRADE CASE tools [13]. Thus, completely formal and exact business modeling will also be available in these tools. Certainly, several pragmatic extensions of the described language are necessary for this purpose.

REFERENCES

1. A.L.Scherr, "A new Approach to Business Processes", IBM Systems Journal, Vol.32, No.1, 1993, p.80-97.
2. A.G.Nilsson, "Business Modeling as a Base for Information System Development", 3rd International Conference on Information Systems Developers Workbench, Gdansk, 1993.
3. R.Gustas, "From Conceptual to Business Modeling", Proceedings of the Baltic Workshop on National Infrastructure Databases, Vol.1, Trakai, Lithuania, 1994, p.218-229.
4. A.Aue, M.Breu, "Distributed Information Systems: An Advanced Methodology", IEEE Transactions on Software Engineering, Vol.20, No.8, 1994, 594-605.
5. M.Grasmanis, I.Medvedis, "Approach to Behavior Specification and Automated Test Generation for Telephone Exchange Systems", Proc. Fifth SDL Forum, North-Holland, 1991, 291-302.
6. P.McBrien, N.Niezette, D.Pantazis, A.H.Seltveit, U.Sundin, C.Theodoulidis, G.Tziallas, R.Wohed, "A Rule Language to Capture and Model Business Policy Specifications", CAiSE 91, Trondheim, Norway: Springer Verlag, 1991, 307-316.
7. P.Loucopoulos, B.Wangler, P.McBrien, F.Schumacker, B.Theodoulidis, V.Kopanas, "Integrating Database Technology Rule Based Systems and Temporal Reasoning for Effective Information Systems: The TEMPORA Paradigm", Journal of Information Systems, Vol.1, No.1, 1991, 129-152.
8. C.Theodoulidis, P.Loucopoulus, B.Wangler, "The Entity Relationship Time Model and the Conceptual Rule Language", Swedish Institute for System Development, SISU Report 1992:01.
9. B.Wangler, U.Wingstedt, R.Wohed, "Experience from Rule-based Modeling at Swedish Post", Proceedings of the Baltic Workshop on National Infrastructure Databases, Vol.2, Trakai, Lithuania, 1994, 35-49.

10. G.Harhalakis et. al., "Implementation of Rule-Based Information Systems for Integrated Manufacturing", IEEE Transactions on Knowledge and Data Engineering, Vol.6, No.6, 1994, p.892-908.

11. G.Held (Hrsg), "GRAPES Language Description: Syntax, Semantics and Grammar of GRAPES", Siemens, 1991.

12. "GRADE V1.0: Modeling and Development Environment for GRAPES-86 and GRAPES/4GL: Language Description", Siemens Nixdorf, 1993.

13. J.Barzdiņš, A.Kalnins, K.Podnieks et.al., "GRADE Windows: an Integrated CASE Tool for Information System Development", Proceedings of the SEKE'94, Knowledge Systems Institute, 1994, p.54-61.