Library

Library loan and photocopying permit

Title of Thesis: Specification Issues & Verification in a Pascal-like Language

Candidate: Hayri SAYI

Three copies of the above thesis are now formally submitted for the degree of M.Phil.... Should the Senate of the University award me either this degree or another higher degree in respect of this thesis, I hereby agree that from the date of the award + the thesis may be made available for inter-library loan or for photocopying and that the abstract may be made available for publication.

+ (or exceptionally the time stipulated under the proviso to reg. 20, viz

Signature: [redacted] Date: 7th October 1982

SOUTHAMPTON UNIVERSITY LIBRARY

Date ~~of Issue~~ due for return

RETURNED 1 7 FEB 87

MAR 87

01. MAR 90

UNIVERSITY OF SOUTHAMPTON

FACULTY OF ENGINEERING AND APPLIED SCIENCE

DEPARTMENT OF ELECTRONICS

SPECIFICATION ISSUES AND VERIFICATION IN A PASCAL-LIKE LANGUAGE

by

Hayri Sayı

Thesis submitted for the Degree of Master of Philosophy

- 1983 -

ACKNOWLEDGEMENTS

First of all, I wish to express my gratitude to Dr. Bernard A. Carré whose constant assistance has been determinant in solving the problems proper to the research area, as well as those of an administrative nature, throughout the past two years.

I must thank the British Gas Corporation for supporting me financially during this period.

The helpful assistance of Mrs J. Wright-Green in the typing of this thesis is also gratefully acknowledged.

CONTENTS

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND APPLIED SCIENCE

ELECTRONICS

Master of Philosophy

SPECIFICATION ISSUES AND VERIFICATION IN A PASCAL-LIKE LANGUAGE

by Hayri SAYI

The effectiveness of very complex, expensive and highly sensitive computer applications depends largely on the correctness of the software in use.

We have tried in this thesis to emphasise the rôle of the specifications as a first step in the design of the verifiable software products. Two techniques for writing formal specifications are described. One of them constructs an abstract state-machine, and the other defines an algebra by means of axioms. Extending an implementation language to accommodate specifications has also proven to be very useful to the verification process, giving birth to languages such as Gypsy and Euclid, both based on Pascal.

A sub-set of Pascal, called Pascal-Minus, was chosen, and a translator from it into the Functional Description Language (FDL) of the Department was developed, to check the conformity of the programs written in Pascal-Minus in relation to their specification in the form of Boolean expressions, using the existing facilities in the Department.

An example is given to illustrate the use and the capabilities of the system which can be extended to incorporate other control and specification constructs, thus increasing its power of expression.

1. INTRODUCTION

Ability to demonstrate in advance the correctness of a design is an important criterion for a discipline in order to be considered as a science. To day, nobody would start building a bridge before its design has been formally proven safe. But, this is more or less the way in which software production goes nowadays. Very expensive and/or highly sensitive applications, whose complexity altogether largely surpasses the grasp of human intellect, are put into use before formally proving that their actual behaviour will meet the intents of the designers. The first American space probe to Venus (Mariner I) which had strayed from its original trajectory in June 1962 and had to be destroyed because of an error in one of the guidance programs in its onboard computer, is such an example. Another major software failure which could end up in a nuclear holocaust was that of early warning systems controlling nuclear missiles. (For a detailed review, see [GER76].)

However, in the past decade or so, considerable effort has been devoted to the development of techniques for the systematic design of well-structured software. The term "structured programming" was first coined by E. W. Dijkstra [DIJ72] to express a methodical attitude towards programming effort, that is the admission of the limitations of our power of comprehension at any one moment. This recognition can be used to our advantage if we divide the problem in hand into independently treatable parts [WIR71, WIR73]. But, beside the unobvious task of recognising subproblems in a large system, each of these subproblems and/or their interrelations can still be too complex to be mastered totally.

The measure of the complexity is the amount of information to be apprehended at any one moment, and this amount can be reduced via abstraction. Thus, by separating those attributes that are relevant in a given decomposition of the task from those that are not, we can end up with intellectually manageable subtasks that we will call modules.

After having emphasised the importance of program verification, and examined the steps taken in that direction by means of modularity and abstraction, we can now look into the whole process of software development in this perspective.

At the beginning of the process is the concept we want our software to implement. This concept can originate from our intuitive understanding of a problem or can be conveyed to us by means of an informal description. There can be many programs to implement it correctly, but their correctness can be stated only in informal terms. This is well away from proving formally that the implementation meets the original concept. What we need is a formal description of the concept which can be analysed for consistency, completeness and conformance with intuition. This formal description will be called a specification, and once it is proved that it captures the original concept correctly, the latter can be abandoned, taking us onto firmer ground than before, vis-a-vis program verification.

Obviously, through systematic decomposition and abstraction, a modular specification will be produced ; giving way to a modular implementation. At this stage, several layers can be inserted between the formal specification and its actual implementation, creating a hierarchy of abstractions.

At each step, the conformance of lower levels with the specifications at the immediately higher level must be proved. Thus, the whole proof issue is divided into several steps, by the systematic application of the methodology in a top-down manner, hiding the lower level implementation information from the higher levels to which it is not relevant.

This program construction methodology is also helpful for the rest of the software development process. The effects of any subsequent change can easily be located, and because of the information hiding inside each module and at each level, only the concerned modules need to be modified. The specifications also provide a good means of documentation, conveying the intents of their

designers in a precise and unambiguous way. Any residual ambiguity in the specifications would be uncovered during the analysis for consistency, completeness and conformance with the high level concept, before starting its implementation. Proofs for each module can be carried out separately, assuming that the behaviour of other modules will conform to their specification. There is no need to wait until the entire program has been coded in order to proceed with the proof. On the contrary, the stepwise refinement process from the higher level specifications towards the lower level implementations is not a linear one [SWA 82]. Each refinement will provide feedback to previous steps, helping them to evolve together. With this constructive approach to the software design ; decomposition, abstraction and their specification issues become a cornerstone to the whole process and the proof that the final product (i. e. executable code) will meet its original concept is incorporated into the whole development process.

The next chapter will present three selected techniques to specify software objects which we found particularly important and promising. One of them consists of writing implicit specifications by describing the states of an abstract (and not necessarily finite) state-machine. The second uses algebraic relations to define abstract data types. The third is an attempt to bridge the gap between specifications and their implementation.

The third chapter presents in detail the Pascal-Minus Project which adopts the third approach of the previous chapter. Pascal-Minus is a sub-set of Pascal which permits insertion of assertions to state formally the intents of the designer for a given implementation. Programs written in Pascal-Minus are translated into the Functional Description Language (FDL) of the Department to check them in relation to the specifications in the form of Boolean assertions.

In the last chapter, an example will be given to illustrate different points made in the preceding chapters.

2. FORMAL SPECIFICATIONS OF DATA ABSTRACTIONS

In this chapter, three specification techniques will be introduced, illustrated with an example problem. One of them uses an abstract state machine model to describe its states and the state transformations that can be accomplished by the application of different operations on it. The second technique defines an algebra by means of axioms to describe the object of interest, thus requiring of its users greater mathematical sophistication than the first one. The third technique consists of extending an implementation language by specification features, which can then be compiled together for both static (compile-time) and dynamic (run-time) checks. A discussion will follow on the respective merits of each method and the problems experienced during their use.

2.1 Description of the example problem

The user's requirements of a system are often quite difficult to formulate formally. They are usually described informally in a natural language, and can be unnecessarily wordy by going into the implementation details or incomplete by omitting sensitive information about the expected behaviour of the system under some conditions. It is not realistic to expect a text of several hundred pages to be consistent, complete and up-to-date. Even so, its very dimension surpasses the capacity of a human-being to apprehend it in its totality. The necessity for concise, consistent and complete formal specifications is beyond argument. The mere statement of this fact is important, but we still have to find the methodology based on a sound ground in order to be able to proceed to the necessary checks to guarantee the aforementioned properties.

We have taken our example problem from everyday life : How to describe a lift controller. The user will call the lift from a certain floor, will wait for its arrival, will get in and push the button corresponding to the floor he/she wants to get to and will

expect the lift to stop at that floor. But, usually he/she will not be alone in using the lift and other users with different and probably contradictory wishes will interfere. If we want to avoid arguments between passengers about which direction it should got first, or simply make it more efficient both for the user's interest and the economy of energy/time, we should find a fair strategy to control it.

The same problem is encountered in the specification of a disc-handler. [HOA74, ABR79] The strategy, called 'lift algorithm', is to go in the same direction until all demands in this direction are satisfied, and then revert to the opposite direction and so on. In this way, the demands are not satisfied on a first-come first-served basis, but depend on the actual position and moving direction of the system. Obviously, this is a cyclic, non-terminating activity, and we have found the task of formally specif ying the general problem difficult within a fixed discipline.

We will first introduce an algorithm in the form of equations which will form a basis for requirement specifications of the lift controller. We will try to stick to the identifier-names chosen to represent various properties of the system, in order to facilitate the comprehension of the specifications in different techniques. These following identifier names will be underlined.

The lift can be going up, down or stopped. The calls are stored into a mem, indexed by the integer floor numbers, from/to which they are made. The lift is at floor loc. max is the highest floor required, min is the corresponding floor in the opposite direction. spin keeps track of the direction of movement. (True if going upwards, otherwise false) arrived (i) becomes true when the lift gets to the floor i. By using them, we can write the following equations to describe the behaviour of the lift. We can see these equations as guarded commands, so whenever the expression on the left of $\Rightarrow$ evaluates true, commands on the right of $\Rightarrow$ will be executed, irrespective of the order in which they appear. This is an apparently non-deterministic specification but as the guards are disjoint, there will

not be any non-deterministic choice between them at the time of execution. So we can rather see them as independent machines which execute the same instructions whenever their guard evaluates true. There can be only one machine executing at any one moment so that they can freely operate on the same data base without any need for the control of its access. The set of these independent machines forms a deterministic machine altogether. The equations are given in Table 1.

stopped and mem (loc) ⇒ open door ; not mem(loc);
stopped and not mem(loc) and spin and max > loc ⇒ up; increment loc;
stopped and not mem(loc) and spin and max = loc and min < loc ⇒ not spin;
stopped and not mem(loc) and not spin and min < loc ⇒ down; decrement loc;
stopped and not mem(loc) and not spin and min = loc and max > loc ⇒ spin;
not stopped and arrived (loc) and mem(loc) ⇒ stopped;
up and arrived (loc) and not mem(loc) ⇒ increment loc ;
down and arrived (loc) and not mem (loc) ⇒ decrement loc ;
otherwise ⇒ skip;

initially loc = 0 and stopped and spin ;
invariants max ⩾ loc and min ⩽ loc;

Table 1. Lift Controller Specification

2.2 Specifying by a Description of Abstract State Machine :

The ideas introduced in the first chapter about modularity and hierarchy of abstractions in structured programming are put into practice by this specification technique. Modules are machines at different levels in the hierarchy and lower level machines execute the specified function of higher level machines. At the top level is the machine, the behaviour of which we want to specify. As the implementation details are hidden by means of a hierarchy of abstractions, we can only observe the external behaviour of a machine, in other words the denomination of abstract machines.

The external behaviour of a machine can be described by means of V-functions (Value functions) which return the value of a state variable at the moment of their call, and of O-functions (Operation functions) which describe the state transformations by attributing new values to the V-functions in terms of their values before the transformation of the state by an O-function call and the function parameters. The set of possible V-function values of the machine defines its internal state-space and a particular set of values denotes one of its states. Each O-function call is a function, mapping a state of the state-space to another one. This descriptive method was introduced by Parnas. [PAR72, PAR72a, PAR75] .

Each V-function specification contains a comment clause describing its result, a returns clause declaring the result type, an initially clause defining its initial value and an exceptions clause stating conditions under which the call will result with an error notification.

An O-function specification contains a comment clause describing the transformation accomplished by its call, an exceptions clause as before, and an effects clause defining new values for each V-function.

Initially, exceptions and effects clauses are expressed in terms of assertions. The assertions in initially are in terms of module constants. For exceptions, the assertions are written in terms of constants and the values of V-functions before the call. In order to distinguish between the values of V-functions before and after the call, the former is quoted. For effects, the assertions are written in terms of constants and V-function values, by assigning to the unquoted new values of V-functions, an expression containing quoted former values and constants. These assertions must hold upon exit from the O-function.

The specification methodology put forward by Robinson, Levitt, Neumann and Spitzen from the Stanford Research Institute

contains five distinct stages, [ROB77, SPI78, ROB77a, ROU77] three of which use a formal specification language called 'SPECIAL'. We will briefly introduce these five stages.

Stage 1 Each module, with a list of its V- and O-functions, is placed in a hierarchical ordering. The visibility of these functions at higher levels is decided at this stage.

Stage 2 Each module is formally specified as described before. The effects specifications of O-functions can be checked for self-consistency, as an inconsistent module specification will give way to an implementation whose correctness cannot be proved. General properties of a module can be expressed in terms of global assertions which can be used as lemmas to simplify the proof of a program that calls functions of the module. They are written in terms of V-functions of the module. Their proof can be carried out by showing that they are true for the initial values of V-functions and also after any sequence of O-function calls.

Stage 3 Decisions about how to represent the state of level i in terms of the states of level i-1 are made. These will be expressed by defining surjective functions from a subset T' of the state space T of level i-1 to the state space S of level i. There can be several states at level i-1 which can map to a single state at level i. As a single state transformation at level i can be accomplished by a sequence of state transformations at level i-1, not all states of level i-1 have images at level i. These aspects will be illustrated during the specification of the lift controller by this method. V-function values of level i are expressed in terms of V-function values of level i-1. By substituting each V-function reference in the specification of level i by its instantiated mapping function expression, mapped specifications are obtained. These can be proved consistent in the same way as in stage 2. Thus, specifications of level i are transformed into assertions expressed in terms of only V-functions of level i-1. By using specifications of levels i and i-1, and the mapping functions, the correctness of the implementation of level i in terms of level i-1 can be checked. At

this stage several important system properties can be stated and proved before any code is written.

Stage 4 Abstract programs using the functions of level i-1 and the control constructs of a programming language are written to implement each of the functions of level i. A proof of correctness of these abstract programs with respect to the specifications of level i and to the mapping functions between levels i and i-1 must be given. This is accomplished by using an extension of Floyd's method [FLO67] , axiomatizing the generation of verification conditions for programs calling O-functions. The input and output assertions for these abstract programs are derived from the mapped specifications.

Stage 5 Primitive functions of level O and of the abstract programming language used at stage 4 are translated into executable code. The communication mechanism between the levels must also be decided. The end product should behave in the same way as specified at the highest level by user's requirements, irrespective of the hidden behaviour of lower levels.

In this way, the highest level abstractions about "what is to be done?" are separated from the data representation and implementation problems corresponding to "how is this to be done?". The complexity to be dealt with at any level is reduced to an intellectually manageable size. A concise and easily understandable design is obtained, whose properties can be stated, even in the absence of proofs. The proof of a large program is divided into the proofs of several small programs whose properties are locally expressible at each level.

After having described the methodology, we can now apply it to the lift controller problem. Only the specification issues will be dealt with (Stages 1 to 3).

The state of the lift controller will be completely determined

by assigning a value to each of its state variables. These represent the location of the lift (loc), its status, i.e. going up, going down or stopped (move), the position of the door, i.e. closed or open, waiting calls (mem) and its next status (spin). Table 2 gives the specifications for the lift controller.

The initial state of the lift is described by the initially clause for each of its state variables. We can see that the lift is stopped at the ground floor with its door closed, and there is no waiting call.

The effects of a call, of its subsequent arrival to a new floor and of the opening of its door are described by the effects clause of the state transformations call, arrived and next respectively.

type Lift Controller = module

V-function loc
returns integer
initially $\emptyset$
comment returns the level of the lift
exceptions none

V-function move
returns {up, down, stopped}
initially stopped
comment returns the status of the lift
exceptions none

V-functions door
returns {open, closed}
initially closed
comment returns the status of the lift-door
exceptions none

V-function mem (i:integer)

returns boolean

initially $\forall$ k (mem(k) = false)

comment returns true if floor i has been called, false otherwise

exceptions none

V-function spin

returns {up, down, stopped}

initially stopped

comment returns the next status of the lift

exceptions none

O-function call (i:integer)

comment insertion of a call for floor i

effects loc = 'loc'

$\forall$ k (mem(k) = if k = i then true else 'mem'(k))

move = if 'spin' = stopped then
if i > 'loc' then up
else if i < 'loc' then down
else stopped
else 'move'

door = if 'spin' = stopped and i = 'loc' then open
else closed

spin = if 'spin' = stopped then
if i > 'loc' then up
else if i < 'loc' then down
else stopped
else 'spin'

O-function arrived (i:integer)

comment hardware signalling of the arrival to the floor i

effects loc = i

O-function arrived (i:integer) continued/....

$\forall k$ (mem (k) = if k = i then false else 'mem' (k))
move = if 'mem' (i) then stopped else 'move'
door = if 'mem' (i) then open else closed
spin = if 'mem' (i) then

if 'move' = up and $\exists k:(k > i$ and 'mem'(k))
then up
else
if $\exists k:(k < i$ and 'mem'(k)) then down
else
if $\exists k:(k > i$ and 'mem' (k)) then up
else stopped
else 'spin'

O-function next

comment 16s. after the opening of the door, this function is automatically called to close the door and assign the next status of the lift.

effects loc = 'loc'
$\forall k$ (mem(k) = if k = loc then false else 'mem' (k))
move = 'spin'
door = closed
spin = 'spin'

Table 2. Specifications for the lift-controller

At this stage, we can write global assertions to express the general properties of the lift-controller, and prove them. As a security condition, the door should be closed while the lift is moving. In terms of state variables at this level, that gives :
$\lceil$(move $\neq$ stopped) and (door = open)$\rceil$ = false. If we show that this assertion is true for the initial values of V-functions and that any sequence of O-function calls respects its truthfulness, we can

be assured that this global assertion will never be violated during the whole life-time of the specified module. Initially, move = stopped and door = closed, so the assertion is true. After a transformation by the O-function call, if the door is open, then $\lceil$ ('spin' = stopped) and (i = 'loc') $\rceil$ must be true, in which case move equals stopped, so the assertion is again true. After a transformation by the O-function arrived, if the door is open, then 'mem' (i) must be true, in which case move equals stopped. Finally, after a transformation by the O-function next, the door is closed. Now, we can be sure that in any conform implementation of these specifications, never will the door be open while the lift is moving.

We can also prove that, if the floor at which the lift is, has been called, then the lift must stop. This can be expressed as : $\lceil$ (move $\neq$ stopped) and mem (loc) $\rceil$ = false, which again is true for the initial state, and any O-function call which finds it true, leaves it also true upon the execution of its transformation. In the same way, if there is a floor, other than that at which the lift is, which has been called, then the lift must move at its next state. This gives $\lceil$ (i $\neq$ loc) $\wedge$ mem (i) $\wedge$ (spin = stopped) $\rceil$ = false.

These three global properties give us enough confidence about the security and the fairness of the design, even before any line of code has been written. This is a very positive point about the method.

In order to illustrate the point about the mapped specifications which were mentioned in the description of the third stage of the method, we will go one level further down to give a specification of the memory as a lower level module. At this lowest level, we will hide the information about the number of floors serviced by the lift. This additional information will necessitate the insertion of exceptions about the bounds of the memory. Table 3 gives the specification of the module Memory.

With the introduction of this module at level 1, the values of the V-function mem at level 2 can be expressed in terms of the V-function read at level 1. By substituting these new expressions into the O-function specifications at level 2, we obtain the mapped specifications. In the O-function call specification, the new value of the V-function mem will be expressed by :

$\forall$ k (read (k) = effects of write (i, true)).

In the same way, in the O-function arrived, the new value of the V-function mem will be :

$\forall$ k (read (k) = effects-of write (i, false)).

That is, the effect of the transformation expressed by the O-function arrived, at level 2, on the value of the V-function mem, is implemented at level 1 by that of the O-function write on the value of the V-function read. The mapping function at the stage 4 of the implementation will be : mem (i) = read (i).

At this stage, we can again prove general properties of the lift controller, by using the supplementary information made available at this lowest level. As the exceptions clause expresses it, the lift is not allowed to go under the ground floor, or over the highest floor (10 at this example), so we can prove that : $(0 \ll loc \ll N)$ = true all the time.

```
type Memory = module
    N:integer = 10
    V-function read (i:integer)
        returns boolean
        initially ∀k (read (k) = if 0≤i≤N then false
                                 else undefined
        comment returns value of ith element in memory
        exceptions bounds : i < 0 or i > N
```

O-function write (i:integer, j:boolean)
comment sets value of i^{th} element in memory to j
exceptions bounds : i $<$ 0 or i $>$ N
effects $\forall$k (read (k) = if k = i then j
else 'read' (k))

Table 3. Specifications of the module Memory

2.3 Algebraic Specifications

The idea of binding together the storage structures representing a type and its operations is not a new one. The class construct of SIMULA 67 [DAH70] denotes a collection of such operations, without offering any representation - independent means for specifying the effect of the operations. An abstract data type specification is a unit which explicitly gives the properties of its values and operations. A relatively large class of implementations are then possible. In this way, the specifications contain only the explicit relationships among the operations without going into the representation details.

Basically, one of the two main approaches has been adopted by different groups of researchers, although notational variations can exist within the same approach. One of these approaches is the axiomatic specifications of Hoare [HOA69, HOA72] which found its expression in the design of the programming language EUCLID [LAM77, LON78]. The second approach is the algebraic specifications method introduced by Guttag [GUT75, GUT77, GUT78, GUT78a, GUT80, GOG78, LIS75]. An interactive system for specification and verification has been developed at the Information Sciences Institute of the University of Southern California. This system, called AFFIRM, checks data type specifications for consistency and completeness and carries out proofs based on these specifications [MUS77, MUS80, MUS80a, GER79]. In the following, we will describe this second approach.

An algebraic specification of an abstract type consists of three parts, of which, the first two, giving respectively syntactic and semantic specifications, are compulsory. The third part deals with restrictions, if there are any exception conditions.

The syntactic part gives the names, domains and ranges of the operations associated with the type. Each operation, whose range differs from the type of interest (TOI) is called an observer. Between the operations whose ranges are TOI, we can distinguish those which construct new values of the TOI (called constructors) and others (called extensions) whose result can be expressed in terms of constructors.

The semantic part is a set of axioms which defines the meaning of operations by stating the effect of a constructor on the values of the observers and the extensions. Only free variables, if - then - else expressions, boolean expressions and recursion can be found on the right hand side of the equations. In terms of the abstract state-machine vocabulary, constructors are O-functions, observers are V-functions, and extensions are O-functions whose effect can be expressed in terms of other O-functions. An example of algebraic specification is given in Table 4. for a set of integers not greater than 255.

type Set [integer]

syntax

new set :		→	Set
insert :	Set X Integer	→	Set
has? :	Set X Integer	→	Boolean
remove :	Set X Integer	→	Set

semantics

declare s : Set ; i, i' : Integer

(1) has? (newset, i) = false

(2) has? (insert (s, i), i') = if i = i' then true
else has? (s, i')

continued overleaf/...

(3) remove (newset, i) = newset

(4) remove (insert (s, i), i') = if i = i' then remove (s, i')
else insert (remove (s, i'), i)

restrictions

$i > 255 \Rightarrow$ failure (insert, i)

Table 4. Algebraic Specifications for a set of integers not greater than 255.

We can immediately see from the example that newset and insert are constructors, has? is an observer and remove is an extension. Any value of the TOI can be constructed starting from newset and only by the application of a sequence of insert operators. As this type is designed to denote sets which contain only integers not greater than 255, any attempt to insert an integer greater than 255 will fail. The restrictions clause requires from the implementor to notify this failure.

An algebraic specification of the lift controller has been written (see Table 5) to enable us to compare this method with the previous abstract state-machine method. Two new observers, max and min have been introduced to keep track of the highest and lowest floors inquired respectively. In this specification, new, call and arrived are constructors ; loc, mem, move, spin, max and min are observers ; and next is an extension (all the values of type Lift can be constructed without using the next operator). An asterisk appears before arrived and next in the syntactic specifications to indicate that these are hidden operators which cannot be called by the users of the type Lift. Their effect is as stated in Table 2 and does not concern the users of the type Lift.

We will prove by data induction a property of the type

Lift as specified in Table 5. This property was assumed to be correct when we wrote the specifications, so its proof will not only increase our confidence in these specifications, but will also facilitate its comprehension by the readers who do not know that this assumption was made.

type Lift

syntax

new :		→ Lift
call :	Lift X Floor	→ Lift
*arrived:	Lift X Floor	→ Lift
loc :	Lift	→ Floor
mem:	Lift X Floor	→ {true, false}
move:	Lift	→ {up, down, stopped}
max:	Lift	→ Floor
min :	Lift	→ Floor
spin:	Lift	→ {true, false}
*next:	Lift	→ Lift

semantics

declare l: Lift ; f_1, f_2 : Floor

(1) loc (new) = $\emptyset$
(2) loc (call (l, f_1)) = loc (l)
(3) loc (arrived (l, f_1)) = f_1
(4) mem (new, f_1) = false
(5) mem (call (l, f_1), f_2) = if f_2 = f_1 then true else mem (l, f_2)
(6) mem (arrived (l, f_1), f_2) = if f_2 = f_1 then false else mem (l, f_2)
(7) max (new) = -1
(8) max (call (l, f_1)) = if $f_1 >$ max (l) then f_1 else max (l)
(9) max (arrived (l, f_1) = if f_1 = max (l) then f_1 - 1 else max (l)
(10) min (new) = +1
(11) min (call (l, f_1)) = if $f_1 <$ min (l) then f_1 else min (l)
(12) min (arrived (l, f_1)) = if f_1 = min (l) then f_1 + 1 else min (l)

contd overleaf/..

(13) spin (new) = true

(14) spin (call (l, f_1)) = if min (l) $>$ max (l) then
if $f_1 \geq$ loc (l) then true
else false
else spin (l)

(15) spin (arrived (l, f_1)) = if mem (l, f_1) then
if max (l) = f_1 and min (l) $<$ f_1 then false
else
if min (l) = f_1 and max (l) $>$ f_1 then true
else spin (l)
else spin (l)

(16) move (new) = stopped

(17) move (call (l, f_1)) = if min (l) $>$ max (l) then
if $f_1 >$ loc (l) then up
else
if $f_1 <$ loc (l) then down
else stopped
else move (l)

(18) move (arrived (l, f_1)) = if mem (l, f_1) then stopped
else move (l)

(19) next (new) = new

(20) next (call (l, f_1)) = call (l, f_1)

(21) next (arrived (l, f_1)) = if mem (l, f_1) then
if spin (arrived (l, f_1)) and max (l) $>$ f_1 then
call (l, max (l))
else
if min (l) $<$ f_1 then call (l, min (l))
else arrived (l, f_1)
else l

Table 5. Algebraic specifications for a Lift Controller

The property to be proved is :

$$\left[\text{min (l)} > \text{max (l)}\right] \equiv \forall f_1 \, (\text{mem (l, } f_1\text{)} = \text{false})$$

It must be true for the constructors which do not take any argument. The only operator of this kind is new for which max (new) = -1, min (new) = +1 and mem (new, f_1) = false, so the equivalence relation is true. If we assume it to hold before an execution of the operation call, we must prove that it will still hold afterwards. The axiom 5 tells us that after call there will at least be one f_1 such that mem (1, f_1) will be true. There can be four cases for min and max :

Before call — After call

(a) $f_1 \geqslant \min(1) \wedge f_1 > \max(1) \Rightarrow$ { max (1) = f_1 (Axiom 8); min (1) unchanged (Axiom 11)

(b) $\min(1) > f_1 > \max(1) \Rightarrow$ { max (1) = f_1 (Axiom 8); min (1) = f_1 (Axiom 11)

(c) $f_1 < \min(1) \wedge \max(1) \geqslant f_1 \Rightarrow$ { max (1) unchanged (Axiom 8); min (1) = f_1 (Axiom 11)

(d) $f_1 \leqslant \max(1) \wedge f_1 \geqslant \min(1) \Rightarrow$ { max (1) unchanged (Axiom 8); min (1) unchanged (Axiom 11)

In all cases, we will end up with max (1) $\geqslant$ min (1), so that both sides of the equivalence relation will evaluate false. The equivalence is conserved by the application of the operation call.

We have to repeat the same reasoning for the last constructor arrived, which is a hidden operator. It is applied whenever the lift reaches a new floor on its movement, and we can see from the axiom 17 that the lift can only be put into movement by the application of call which introduces a new value true into mem. So we know that there is at least one f_1 such that mem (1, f_1) will be true before the application of arrived. As we assume that the equivalence relation holds before the application of arrived, we can say that max (1) $\geqslant$ min (1) will be true at that time. It can be proved that the case max (1) = min (1) denotes the situation in which there is

just one element true in mem corresponding to the next coming floor. In this case, after the application of arrived :

$$f_1 = \max(1) = \min(1) \Rightarrow \begin{cases} \max(1) = f_1 - 1 & \text{(Axiom 9)} \\ \min(1) = f_1 + 1 & \text{(Axiom 12)} \end{cases}$$

As the only element of mem which was true before the application, will become false (Axiom 6) ; $[\min(1) > \max(1)] \wedge \forall f_1 \quad (\text{mem}(1, f_1) = \text{false})$ will hold after, keeping the equivalence relation true. It can also be proved that the case $\max(1) > \min(1)$ denotes the situation in which there is an element true in mem which does not correspond to the next-coming floor. In this case, the application of arrived will leave that element true (Axiom 6), so we have to prove that the other side of the equivalence relation is also false :

Before arrived		After arrived	
$\max(1) > \min(1)$	$\Rightarrow$	$\max(1) > \min(1)$	(Axioms 9, 12)

The only effect of arrived on max and min can be incrementing min or decrementing max so that the relation "greater than" can only be transformed into "greater or equal" which concludes the proof.

2.4 Programming Languages with Specification Facilities

The ability to describe the behaviour of a software product in terms of an abstract object, and to demonstrate its global properties by using only the formal specifications of that object, is a constructive approach to the whole software production process, with a deliberate emphasis on the verification issues. However, the ' gap ' between the formal specifications and the final executable code can hamper the expected benefits of the approach, and all our efforts for producing and verifying the formal specifications can be put in jeopardy.

To avoid this pitfall, one can take a programming language and extend it, by incorporating into it the necessary features to the expression of specifications. This approach has

given birth to GYPSY [AMB77], EUCLID [LAM77, POP77, LON78, HOL78, WOR79, WOR81] and SP-EUCLID [AND78], to mention only three of them. They are both based on PASCAL and allow for the gradual refining of the specifications into an implementation to be expressed in the framework of the language.

The verification being the main concern for both of these projects, features of Pascal which made the verification difficult without adding much to its power of expression, were deleted altogether. These include functions and procedures as parameters, labels, GOTO statements, real numbers and, variable parameters for functions which now become pure mathematical functions without any side-effect. While-do and repeat-until loops of Pascal are replaced by a more general loop construct with specific exit points. In the following, we will concentrate on Euclid and give a specified implementation of the lift controller written in it.

A five-man committee was commissioned in 1976 to make minimal changes and extensions to Pascal in order to obtain a verifiable system programming language, with the proclaimed aim of "transferring more and more of the work of producing a correct program, and verifying that it is consistent with its specification, from the programmer and the verifier (human or mechanical) to the language and its compiler" [POP77]. This gave birth to Euclid.

The main atomic unit in Euclid is a module which brings together related types, variables and routines (procedures and functions) with initialization and finalization components that are executed whenever instances of the module are created or destroyed. In that respect, they can be seen as new-type-constructors with the advantages of abstract data types. The dual aspect of this is the information hiding achieved within a module, which enables a system-designer to build a hierarchical structure based on them. We will discuss that aspect more in detail later.

Each module, which has its own local variables, types and routines declared within its body, can only use a name declared in another module if that name appears explicitly in its import-list as well as in the export-list of the module to which it belongs. If a variable name is imported (respectively exported) only for referencing without changing its value, a readonly clause must precede it in the import-list (respectively export-list). A new assignment to a variable outside its own module can only be made, if and only if, its name appears, preceded with a var clause, both in the import-list of the module which wants to change its value, and in the export-list of the module to which it belongs.

The same rules apply to the routines (procedures or functions), which can only import names local to their module or already imported into it ; with the exception that the functions cannot import variables. Together with the restriction on variable parameters, the Euclid functions behave like mathematical functions. Thus, they can only have parameters preceded with a const clause in the parameter-list, or can only import names preceded with a readonly clause in their import-list. Procedures can have both call-by-value parameters (preceded with a const clause as for the functions) and call-by-reference parameters (preceded with a var clause in their parameter-list). They can also reference a name by importing it readonly or change its value by importing it var.

In this way, the interface between the modules becomes explicit and the conformance of the module bodies to these import-export rules can be checked at compile-time. For array indices within bounds and variant records, these checks will usually depend on dynamic information, although the compiler can often use declared ranges or flow analysis to do partial checking. In these instances, the Euclid-compiler will generate legality assertions which must all be verified for the program to be legal, i.e. consistent with the language specification, with a defined meaning during the execution. These assertions take the form of Boolean expressions.

The explicit control over the visibility of names in modules and routines is a very important issue, both at the design stage, to see directly the relationships between modules and routines ; at the implementation stage, to hide the implementation details by not exporting them ; and at the maintenance-modifications stage, to locate easily the effects of a change in a module or routine on the other modules or routines. Although Euclid programs may tend to be longer on average than equivalent programs in other languages and thus take longer to write, the extra information supplied allows the Euclid compiler to do a much more comprehensive check, and provides a good means of documentation.

As we had stated at the beginning of this section, verification is the main concern behind the design of Euclid. Therefore, the language has syntactic means for including specifications and intermediate assertions. In Euclid, an assertion is usually a Boolean expression, which is evaluated when the execution, reaches the point at which it appears. If the expression evaluates 'true', then execution proceeds, otherwise a run-time halt occurs. Assertions written in a richer language, containing, for example, quantifiers and specification routines, can be bracketted as comments, to be submitted directly to the verifier. These will be proved either manually or mechanically by using the axiomatic method of Floyd-Hoare [FLO67, HOA69, HOA73] . The proof rules for Euclid are given in [LON78] .

Routines are specified by pre and post-assertions . The pre-assertion must evaluate 'true' at the point the routine is called, and the post-assertion at the point of return.

Modules are specified by a pre-assertion, an invariant-assertion, an abstraction function and specifications for exported routines and types. The invariant-assertion of a module must evaluate 'true' whenever an exported routine of the module is called and whenever it returns, thus maintaining the data integrity of the module.

Moreover, assertions may be placed at any point in the flow of control, to express intermediate properties or loop invariants.

There is no provision for exception-handling in Euclid, as a verified program is expected not to cause a run-time software error, and recovery from unanticipated hardware failures, a non-trivial task. Anticipated conditions must be dealt with using the normal constructs of the language.

In Euclid, a type declaration may have formal parameters, thus making the relationship among similar types to be made explicit in the program. Variant record definitions will use this facility, with the tag being one of the formal parameters. A module is a type constructor, several instances of which can be created by declaring names of that type. And finally, a type or routine declaration can be made visible in the whole module, by prefixing it with a pervasive clause. Names with a pervasive declaration need not be imported into other routines of the module.

In order to illustrate the specification means of Euclid, a specified implementation of the lift controller is given in Table 6. We can see that only the procedures Asked and Arrived are visible outside the module, which constitute the only interface with the 'outside world', i. e. users and hardware. Whenever a new instance of type lift is created, all its local variables are initialised and it will wait in that initial state until a call is made from outside to the procedure Asked, to require it to go to a specific floor and open the doors. While it is moving, other calls can be received together with hardware signals (i. e. calls to the procedure Arrived) warning it of an arrival at the next floor in the direction of the movement. Two type definitions (i. e. Floor Type and Movement Type) are exported too, to avoid redundant declarations of the same type outside the module.

```
const number Of Floors :=10;
var Lift:
      module
      imports (number Of Floors);
      exports (Asked, Arrived, Floor Type, Movement Type);
      invariant ((spin=stopped ⇒ move=stopped) and
                 (move ≠ stopped ⇒ doors = closed) and
                 (move=up ⇒ max > loc) and
                 (move=down ⇒ min < loc));
      pervasive type Floor Type = O.. number of Floors ;
      pervasive type Movement Type =(up, down, stopped);
      pervasive type Door Type = (open, closed);
      var loc, max, min:Floor Type ;
      var move, spin:Movement Type ;
      var doors : Door Type ;
      var mem:array Floor Type of Boolean;

      pervasive procedure Open Doors=
      imports (var doors, var mem, var move, readonly spin, readonly loc);
      pre(move = stopped and mem(loc) and doors = closed);
      post (not mem (loc) and doors = closed);
      begin
          doors:=open; wait 16s.;doors:=closed;
          mem(loc):=false;move:=spin;
      end Open Doors ;
     procedure Asked (const floor:Floor Type)=
          imports (readonly loc, var move, var spin, var mem, var doors);
          pre(true);
          post (mem(floor));
          begin
          mem(floor):=true;
          if spin=stopped then
          if floor > loc then
          begin
            move:=up;spin:=up;
          end;
```

```
        else
          if floor=loc then Open Doors;
          else
          begin
          move:=down;spin:=down;
          end;
if floor > max then max:=floor;
if floor < min then min:=floor;
end Asked;

procedure Arrived=
      imports (var loc, var move, var spin, var mem, var doors,
                  number Of Floors);
      pre(move ≠ stopped and spin ≠ stopped and doors = closed);
      post (not mem (loc) and doors = closed);
      begin
         if move=up then loc:=loc+1;
         else loc:=loc-1;
         if mem (loc) then
         begin
            move:= stopped;
            if (max=loc) and (min < loc) then spin:=down;
            if (min=loc) and (max > loc) then spin:=up;
            if (max <= loc) and (min > = loc) then spin:=stopped;
            if max = loc then max:= ∅;
            if min = loc then min:=number Of Floors;
            Open Doors;
         end;
      end Arrived;

initially
      imports (var loc, var max, var min, var move, var spin,
                  var doors, var mem, number Of Floors);
      begin
         loc:= ∅; max:= ∅; min:=number Of Floors;
         move:=stopped; spin:=stopped; dooors:=closed;
         for i in Floor Type
            loop
               mem(i):=false;
            end loop;
      end;
end module;
```

Table 6. Lift controller as a Euclid module

As it was anticipated at the beginning of this section, a Euclid module can be seen as an implementation of an abstract data type together with its specification: import, export and parameter lists, type and var declarations of a module, representing a syntactic specification of the abstract data type, and, routine bodies together with initialization and finalization parts giving the semantics of it. This is the 'package' aspect of a module, putting together variable and type declarations and the operations which can be accomplished on them.

At the same time, only a limited number of variables, types and routines of a module are made available through its export-list to the 'outside world', hiding all the implementation details from outside. This aspect has important consequences for both software specification, implementation, verification and maintenance-modifications processes.

At the specification level, Euclid modules make the top-down design approach both possible and natural. The concept to be specified at the highest level can be gradually refined through a hierarchy of modules. The obligation to specify explicitly the interface between modules, through import, export-lists, will force the designer to think carefully about the relationship between modules, even before their implementation has started. In this way, only the objects necessary to the interface will be made visible, abstracting from the implementation-choices about how a particular data object or an operation is going to be expressed in terms of a particular programming language construct. As it was explained in the first chapter, this is the only valid approach to the development of large software systems.

Once the top-down design based on modules has been completed, by stating explicitly the relationship between the modules through import-export lists, and, by specifying the functionality of each module with a pre-assertion, an invariant-assertion, an abstraction-function and specifications for its exported routines using pre- and post-assertions ; separate implementation

of each module and its routines can start. If a difficulty encountered during their implementation forces us to modify the initial structure of the software, the transparency of the interface between modules will facilitate the task by assisting us to locate easily the effects of a change on the overall structure. Implementation of each module can be done separately, by assuming about imported objects that their implementation will conform to their specification.

After a complete implementation in Euclid of all modules and their routines has been obtained, it can be compiled [HOL80] to check for the conformance of the code to the rules about import-export lists as well as to the syntactic definition of the language. Whenever dynamic information is needed, as in array indices within bounds or variant records, the compiler will produce a legality assertion which must be verified afterwards. All pre-, post-invariant- and legality-assertions in the form of Boolean expressions are compiled too, and the code augmented with these assertions can be run for debugging. An assertion, which evaluates 'false' at the point of its execution, causes a run-time halt with a suitable message. Although testing cannot prove the correctness of a program, it can significantly reduce time and energy wasted in looking for proofs of programs still containing bugs. After a reasonable degree of confidence has been gained through debugging, proofs can be carried out by using the axiomatic method of Floyd-Hoare which will be examined in the next chapter.

During the maintenance-modifications phase, the modular structure of Euclid programs, together with their specification, facilitates comprehension and helps in localising the effects of a change. These effects are minimised by the information-hiding inside the modules, and a complete module body can be changed without affecting its environment, if its exported objects conserve the same functionality.

2.5 Discussion

After having introduced three methods to specify, implement and prove software objects, we will discuss their relative merits and deficiencies.

The state-machine method of Parnas builds an abstract object by defining the effects of a state-transformation on the values of its state-variables. This is a non-procedural specification method, defining each operation separately in a static way with a reasonable degree of abstraction, and permitting the proof of global properties of the specified object before its implementation. The effects of a transformation are expressed in terms of simultaneous (possibly conditional) assignments to the state variables, leaving out all iteration and recursion possibilities, and thus necessitating introduction of several operations to express a succession of transformations.

In 1978, an attempt was made, by the TRW Defense and Space Systems Group, to extend Euclid to accommodate the state-machine method, which gave birth to SP-EUCLID. This attempt was motivated by the necessity to verify that the operating system developed for the Defense Advanced Research Projects Agency (DARPA) was meeting Department of Defence security requirements. A more detailed analysis of the approach can be found in [AND78].

Another attempt, by the Federal Systems Division of the IBM Corporation, to use the concept of a state-machine as a basis for specifying modules, was reported in [SHA82].

Although, the degree of abstraction and formalism, achieved by the non-constructive approach of the algebraic method, is higher than the one achieved by the first method, it is usually very difficult to find the characterising operations and to express

the relations between them in the algebraic model. Our own experience, and discussions with Dr. Guttag, have shown us that sooner or later one feels compelled to reason in terms of a model other than that of the algebraic specifications.

As the attempt to prove certain general properties of the specified model has shown us, the data induction of the algebraic method is much longer than proving global assertions in the state-machine model. On the other hand, tools can be developed to assist in the design, implementation and verification of algebraic specifications.

It also emerges from the discussions in the literature that algebraic specifications are not appropriate for specifying all possible tasks. For example, it might be impossible to find a finite representation of a type by operations and relations between them [MAJ77].

In all modesty, one can say that, for a given problem, a specification technique can prove to be less appropriate than the others. It is obvious for the example problem of this chapter that the description given in Table 1 is the simplest and the shortest of all, and therefore the easiest to comprehend. But the same formalism could be completely inadequate for a different problem. Although it is very tempting to 'compare' different techniques by applying them to the same problem, their respective evaluation should not exclusively be based upon that restricted experience.

The approach of unifying the means of specification and implementation, in the same notation, is obviously less formal than the others. But, it has the no less obvious advantage of bridging the gap between the formal specifications and their implementation by enabling us to express them both under the same formalism. They are also easier to write and to understand than in the first two methods. The possibility of using the compiler to do important checks about module interfaces cannot be under-

estimated. These reasons made possible the design, implementation and checking of a large piece of software (about 60000 source lines) in Euclid [WOR81]. This is to compare with the Delta Experiment in the algebraic method (roughly 1000 lines) which took 6-8 months [GER79].

The module construct of Euclid can be used both to create instances of abstract data types and to assist us in the design of large software systems in a top-down manner.

The advantages of unifying specification and implementation notations pushed us towards the third approach. The existence of a system capable of assisting in the verification process was also an important factor. These were the main reasons behind the Project Pascal-Minus which will be presented in the next chapter.

As for Gypsy and Euclid, we have chosen Pascal to start with, and eliminated features such as, procedures and functions as parameters, labels and GOTO statements, and real numbers which made the verification difficult.

The idea behind this project is to write specified implementations as in Euclid and to translate them into the Functional Description Language (FDL) to carry on the necessary checks about the conformance of the implementation in relation to the specifications written in the form of Boolean expressions. The next chapter will present in detail our approach.

3. PASCAL-MINUS PROJECT

In this chapter, after having explained our motivations in taking a particular subset of the programming language Pascal, which we call Pascal-Minus, and translating the programs written in that subset of Pascal into the Functional Description Language (FDL) of the analyzer currently being used at the Electronics Department of Southampton University, so that the pre-, post-assertions and the loop invariants introduced into the original Pascal-Minus program texts can be checked by proving the verification conditions generated by the analyzer, we will give a detailed description of the implementation of our translator, together with a precise syntactical definition of both the input and the output languages, i. e. Pascal-Minus and the FDL respectively. We will also discuss the possible extensions to the system so that the initial Pascal-Minus can be enriched to include more powerful control structures, thus enhancing its power of expression and conciseness without adding an excessive burden to the verification process.

3.1 Motivation :

In the preceding chapter, we have introduced three methods of specifying software objects, and discussed of their relative merits and shortcomings. After having written a complete set of consistent specifications, the next step consists of gradually refining them down to the executable code level. At this stage, we have in one hand the specifications which state formally, in one way or another, the original intents of the designer, and in the other hand the implementation in a programming language which is supposed to embody these original intentions. Obviously, the task is to prove that the latter performs what was intended by the former.

For this purpose, two logically equivalent techniques have been suggested and have found a large circulation since their formulation. They both use assertions to state what must be true of the variables at different points of a program. From the point of

view of program execution, these assertions are merely formal comments and no code is generated during the compilation.

The first approach, which is generally called "Floyd-Hoare Axiomatic Logic" [FLO67, HOA69, HOA71, HOA72] consists of writing rules of inference to denote the semantics of the programming language constructs. Beside the fact that these constitute a formal definition of the semantics of the programming language in hand, they can also be used to reason formally about the programs written in that particular language [HOA73].

Given a program with pre-, post-assertions and loop invariants, one can then start from the post-assertion, and using the rule of inference for each language construct, go in the opposite direction of the program execution until one reaches the beginning of the program. After each inference, a condition is generated which must be true at that point of the program, if the post-assertion, which states the expected properties of the program is to hold at the end. These conditions are called 'verification conditions' and the tool which generates them, starting from the post-assertion and using the rules of inference of that particular programming language, is called a 'verification condition generator'.

As an example, we can give the rules of inference for the 'assignment', 'compound', 'if' and 'while' statements of Pascal [HOA73] which are maintained in Pascal-Minus :

Assignment Statements : $P_y^x \{ x:=y \} P$

Compound Statements : $$\frac{P_{i-1} \{ S_i \} P_i \quad \text{for } i = 1 \ldots n}{P_o \{ \underline{\text{begin}}\ S_1; S_2; \ldots; S_n\ \underline{\text{end}} \} P_n}$$

If Statements : $$\frac{P \wedge B \{ S_1 \} Q,\ P \wedge \neg B \{ S_2 \} Q}{P \{ \underline{\text{if}}\ B\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2 \} Q}$$

$$\frac{P \wedge B \{ S \} Q,\ P \wedge \neg B \supset Q}{P \{ \underline{\text{if}}\ B\ \underline{\text{then}}\ S \} Q}$$

While Statements :

$$\frac{P \wedge B \{ S \} P}{P \{ \underline{\text{while}}\ B\ \underline{\text{do}}\ S \} P \wedge \neg B}$$

where : P_i, Q are logical formulas describing properties of data ;
$P\{S\}Q$ is an assertion which expresses that, if P is true before the execution of S, then Q is true after the execution of S. If the execution of S does not terminate, it is also true.

$\frac{H_1, \ldots, H_n}{H}$ is a rule of inference which states that whenever $H_1, \ldots, H_n$ are true assertions, then H is also a true assertion ;

P_y^x means substituting y for all free occurrences of x in P.

If one can prove all the verification conditions, then one can affirm that whenever this program is executed, with initial values for which the pre-assertion is true, and it does terminate, then the post-assertion will be true at the termination.

The second approach, called 'symbolic execution', simulates the execution of a program by maintaining a 'state vector' containing the symbolic values possessed by each program variable. For each path through the program, a 'path condition', which states in terms of symbolic values of program variables the condition under which this path can be executed, is recorded together with the actions accomplished along that path, allowing us to apprehend the input-output relations established by the execution of that program [DAN82].

Depending on the purposes of the analysis, the length of the paths can be fixed in advance or particular paths simulated by a judicious choice of symbolic values for some particularly important variables.

Both approaches are partly mechanizable, taking away from

the humans the tedious and error-prone steps, and leaving them with the important strategic choices about the insertion of assertions at particular points of the program, deciding which paths to follow in particular or proving the verification conditions.

The Analyzer at the Electronics Department of Southampton University adopts the second approach and, given an initial 'state vector' consisting of particular symbolic values for the program variables and a particular path, simulates the execution of that path either statement-by-statement or by using a reduced form of it.

The input language to the Analyzer is the FDL for which we will give a syntactical definition.

Our aim in undertaking the Pascal-Minus Project was to enable us to use the existing tools at the Department to analyse programs written in a sub-set of Pascal, made free from constructs which unnecessarily complicate the verification issues. Certain constructs have definitely been abandoned, others have provisionally been deleted from the initial subset to be added later on, once a skeletal system becomes operational. We shall now proceed to explain our choices which determined the actual form of Pascal-Minus.

3.2 Which Subset of Pascal?

Although Pascal has widely been accepted as a 'clean' programming language, it was not exempt from ambiguous and/or insecure features [HAB73, WIR75, WEL77]. Even its own designer was recommending the deletion of the GOTO statement. Much of the criticisms were directed against functions and procedures as parameters, array bounds, dangling references via pointers, variant records, labels and GOTO statements, and case statements. These criticisms were formulated from the stand-point of the users. When it came to define a programming language based on Pascal, with a priority to the program verification issues, then several language features were deleted to improve verifiability without undue

loss of power of expression. These included functions and procedures as parameters, labels and GOTO statements, real numbers, multi-dimensional arrays and input-output facilities, giving birth to EUCLID [LAM77, POP77, LON78, HOL78, WOR79, WOR81].

In making our choices, we have taken into account these developments and definitely abandoned Pasal features which are not in EUCLID.

On top of that and in order to obtain pure functions without any side-effects, we do not allow Pascal-Minus functions to have call-by-reference parameters. Therefore a parameter list for a Pascal-Minus function cannot contain a 'VAR' clause.

We also wanted to replace each procedure call by its body after having assigned the actual values to the call-by-value parameters, so that the procedural structure of Pascal programs could be transposed into the FDL by 'unwrapping' procedure calls. The same consideration applies to the function calls inside the expressions. Before a statement, containing an expression with a function call in it, is translated, the function body together with the assignment(s) to the function identifier is unwrapped. This obviously necessitates the forbidding of circular calls and recursive functions or procedures, in other words, the call graph of the program to be translated must be acyclic.

To simplify the initial task of building an operational system, certain Pascal features have provisionally been deleted. Repeat-until and for loops are omitted, but the while-do loop is retained. This restriction does obviously not diminish the power of expression of the language. As it will be discussed under the 'extensions' heading, all these constructs will be replaced with a more general loop construct with specific exit points. With and case statements have also been deleted from Pascal-Minus.

As for the predefined types and data-structures, only integer, boolean, character and scalar types are kept. The only constants belong to the defined scalar types. The pre-defined infix Pascal operators, 'or', 'and', 'mod' and 'integer division' are also kept. Appendix I gives the syntax diagrams for Pascal-Minus.

The Pascal-Minus Project consists of building a tree to represent all the declarations. All the attributes of these identifiers are kept in records pointed from the tree. Each procedure or function identifier also possesses a pointer to its parameter list and another pointer to its body which is a linked list of statements. Once this tree is built, the whole or the parts of it can then be translated into the FDL in an interactive way with the user. The part of the translator dealing with that tree is essentially a syntax analyser, very similar in form to other commonly used syntax analysers for Pascal [PEM82, WIR81, AMM81]. Instead of generating P-code or assembly code or any other internal form, it generates a syntax tree which is then traversed to transform it into the FDL form. These points will be explained in detail under the heading 'Description of the Implementation'. Before doing that, we must now describe the FDL.

The FDL contains assignment, if-then-else and GOTO statements with labels. The only predefined types are integer and boolean. Predefined 'or', 'and' infix operators exist. There can only be one program body between 'START' and 'FINISH' clauses.

This last point necessitates the renaming of procedure and function parameters and local variables in order to distinguish them from the global variables, declared at the outermost program level, when the sub-programs will be unwrapped. Labels are unsigned integers.

There exists extensions for Abstract Data Types, but these are not involved in the translation from Pascal-Minus.

In order to illustrate the translation issues, an example

program written in Pascal-Minus and its translated form into the FDL can be found at Figure 1.

3.3 Description of the Implementation :

The implementation is written in standard Pascal and makes just over 1500 lines. The main program PMINUS consists of calls to four procedures : COMPINIT, DECLARATIONPART, BODYPART and QUERY respectively, which we will describe in detail after an introduction to the data structures constructed and used by them. These four procedures are completely independent one from the other and intervene sequentially to perform their task.

COMPINIT initialises the whole system by filling the symbol-buffer SYMBUF, by entering the predefined types 'integer', 'boolean' and 'character', together with two scalar values 'false' and 'true' of the predefined type 'boolean'. It also reads the main program heading and enters the program identifier, so that when the execution of COMPINIT terminates, the system is ready to receive all the declarations, the sub-program bodies and the main program body by calling DECLARATIONPART and BODYPART one after the other.

3.3.1 Data Structures

COMPINIT and DECLARATIONPART build two main data structures. One of them is the tree which keeps record of each identifier together with all its relevant attributes. A main program variable, OUTERBLOCK, points to the root of that tree named IDENTIFIER. IDENTIFIER is a record with the following fields :

Name : Identifier-name in a packed array of eight characters ;

Name 1 : Renamed variable identifier by adding a '#' and an integer in the range of 1 to 99 corresponding

```
PROGRAM TEST2(INPUT,OUTPUT);
   TYPE
     DEVICE=(ACTIVE,PASSIVE);
   VAR
     MODULE:DEVICE;
     TIME:INTEGER;

   FUNCTION TESTMEMORY(MEM:INTEGER):BOOLEAN;
     BEGIN
       IF MEM=1400 THEN TESTMEMORY:=TRUE
       ELSE TESTMEMORY:=FALSE
     END;

   BEGIN {TEST2}
     MODULE:=PASSIVE;
     TIME:=800;
     WHILE TIME<2000 DO
       BEGIN
         TIME:=TIME+1;
         IF TESTMEMORY(TIME) THEN
           BEGIN
             MODULE:=ACTIVE;
             TIME:=2000
           END
       END
  END {TEST2};


'TITLE' PROGRAM TEST2;
'TYPE' CHAR;
'INFIX' MOD(INTEGER,INTEGER):INTEGER;
'INFIX' DIV(INTEGER,INTEGER):INTEGER;
'TYPE' DEVICE;
'CONSTANT' PASSIVE,ACTIVE:DEVICE;
'VARIABLE' MODULE:DEVICE;
'VARIABLE' TESTMEMO##0:BOOLEAN;
'VARIABLE' TIME:INTEGER;
'VARIABLE' MEM#1:INTEGER;

'START'
MODULE:=PASSIVE;
TIME:=800;
1:'IF' NOT ( TIME < 2000 )
'THEN' 'GOTO' 2;
TIME:=TIME+1;
{ BEGIN FUNCTION TESTMEMO }
MEM#1:=TIME;
'IF' NOT ( MEM#1 = 1400 )
'THEN' 'GOTO' 4;
TESTMEMO##0:=TRUE;
'GOTO' 3;
4:TESTMEMO##0:=FALSE;
3:{ END FUNCTION TESTMEMO }
'IF' NOT ( TESTMEMO##0 )
'THEN' 'GOTO' 5;
MODULE:=ACTIVE;
TIME:=2000;
5:'GOTO' 1;
2:'FINISH'
```

Figure 1. A Pascal-Minus program and its translated form into the FDL

to the static declaration level of the variable, a packed array of eleven characters.

Rlink, Llink : Pointers to the immediate right and left neighbours, pointing both to records of type IDENTIFIER in the tree.

Next : Used only for procedure and function identifiers, their parameters and identifiers of an enumerated type, pointing to the rest of the list, in the order they appear in the parameter list or in the enumeration list ; nil for the last identifier or for a function or a procedure without parameter ;

Idtype : Pointer to the second structure which keeps record of the identifier types, nil for a procedure identifier ;

Variant fields :

Konst : Used only for identifiers belonging to an enumerated type, keeps their ordinal number ; { For a declaration, Colour = (Red, Blue, White), Values is 0 for Red, 1 for Blue and 2 for White }

Formalvars, Actualvars : Used for parameter identifiers and for variable identifiers local to a sub-program (i.e. function or procedure), Vlev keeps the static declaration level ;

Proc, Func : Pflev is the static declaration level of the sub-program, and equals the Vlev of all its parameters and local variables ;

Firstvar points to the first parameter identifier in the tree, if there is any, otherwise to the first

declared local variable identifier ;

Bodi points to its body.

The second data structure, named STRUCTURE, is a record keeping the type information. For variables of standard type as 'integer' and 'character', there is a special pointer, INTPTR and CHARPTR respectively, and there is no need for any special entry to that record. For enumerated types, Fconst points to the first element in the enumeration list. Nom keeps the type-identifier in a packed array of eight characters, Noml keeps the renamed type-identifier in a packed array of eleven characters, as for a variable identifier.

Display is a main program variable and plays the rôle of a stack for sub-programs. Whenever a new sub-program declaration is encountered, a new record is pushed into the array. Fname points to the first parameter or local variable identifier of the sub-program. Pname points to the sub-program identifier. Whenever a sub-program body ends, the record corresponding to that sub-program is popped. Top is the index to that array.

Dict is also a main program variable and consists of an array of pointers to a record of type IDENTIFIER. A new element is entered to this array whenever a new sub-program declaration is encountered. DICT[O] points to the main program identifier, the subsequent elements of the array point each to a different sub-program identifier, in the order they appear in the program text. Therefore, there will be just one assignment to each element of the array during the whole lifetime of PMINUS. Level is the index to that array.

Symbuf is a packed array of 1K characters, Symcursor is its index. Symbuf stores the Pascal-Minus text input to the system, in its original form, and each time it is consumed, the procedure Readtext is called to fill it again, until the end of the input text.

Sy of type Symbol is the last symbol read by the procedure Insymbol which recognizes the different tokens of the language. If the last symbol is an operator, Op stores its kind ; if it is an integer, Val stores it ; if it is an identifier or a reserved word of the language, Id of type packed array of eight characters stores it.

Figure 2 shows the representation of names and their attributes for the program in Figure 1.

3.3.2 Building up a Dictionary of Names

DECLARATIONPART builds the whole IDENTIFIER tree. Depending on the existence of the different possible parts in the declaration-list being processed, it will call in turn procedures TYPEDECLARATION, VARDECLARATION or PROCDECLARATION until all the declaration-list is exhausted and a sub-program body or the main program body is encountered. DECLARATIONPART and BODYPART both use the well-known recursive descent technique, found in all one-pass Pascal compilers.

TYPEDECLARATION will enter the new type identifier into the tree, by a call to the procedure ENTERID. As the only user defined types can be of enumerated kind, the procedure SIMPLETYPE will be called to process the enumeration list. Upon termination of the procedure SIMPLETYPE, new type definitions can be processed, if there is any, otherwise TYPEDECLARATION terminates.

SIMPLETYPE will create a record of type STRUCTURE and will enter all the identifiers in the enumeration list into the tree. Fconst of the newly created record will point to the first element of the enumeration list, and the elements of the enumeration list are linked together through next of each element entered to the tree. This procedure has a call-by-reference parameter which returns the pointer to the type information kept in STRUCTURE for the type identifier which has just been processed.

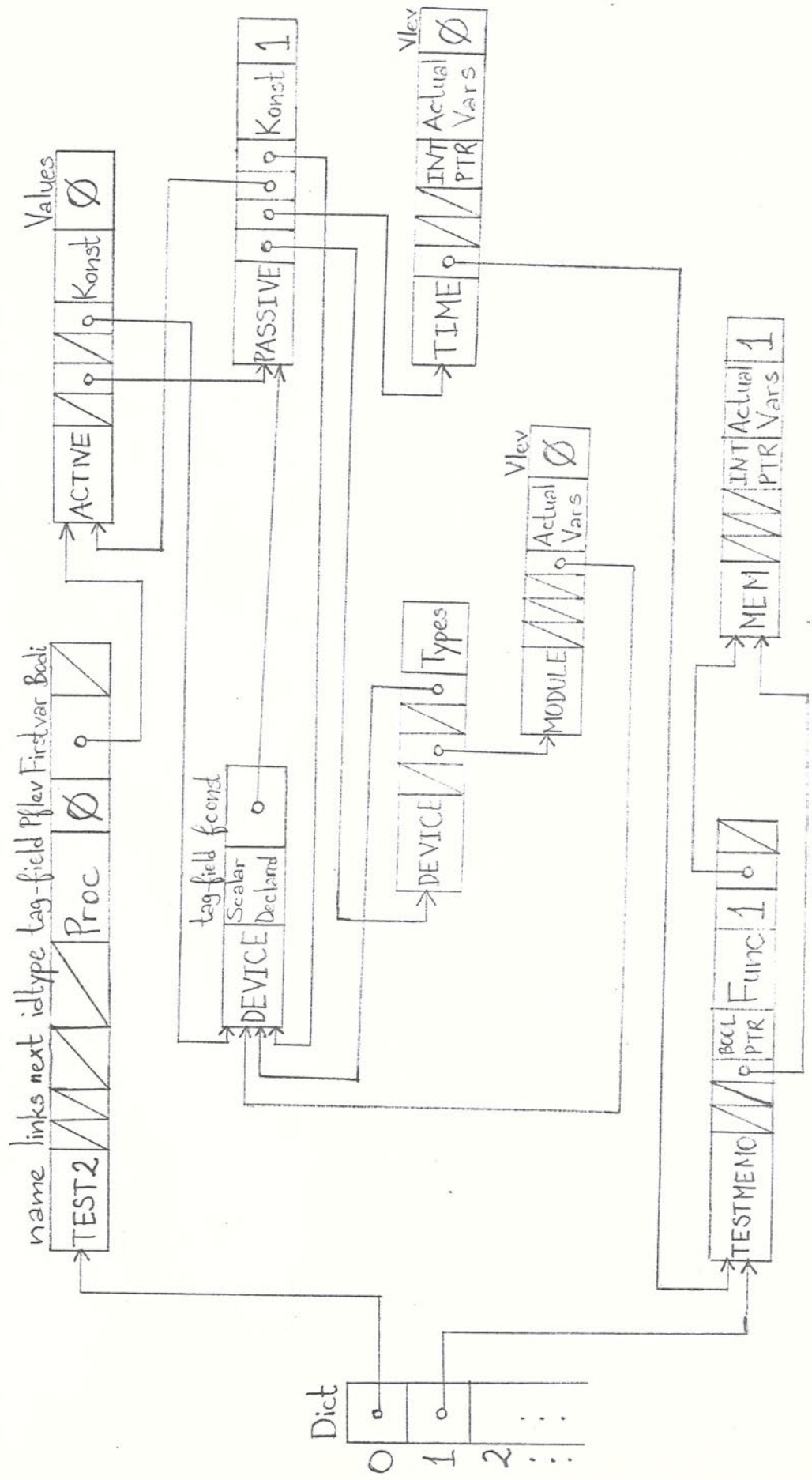


Figure 2. Representation of names and their attributes for the program in Figure 1

VARDECLARATION enters the variable identifier into the tree by calling ENTERID as an actual variable, stores the static declaration level in Vlev, and calls the procedure SIMPLETYPE to get the pointer to the type information corresponding to its type identifier. This procedure will terminate after having processed all the variable identifiers.

PROCDECLARATION will enter, by calling the procedure ENTERID, the sub-program identifier into the tree, as well as into the stack Display and the array Dict. After that, it will call the procedure PARAMETERLIST to process the parameters of the sub-program, and for a function it will call the procedure SEARCHID to find the pointer to the type information corresponding to the result type identifier of the function. This result type pointer will be stored in Idtype corresponding to the function identifier.

PARAMETERLIST will enter the parameters and their respective types into the tree as in VARDECLARATION by calling ENTERID and SEARCHID. On top of that, it will make the difference between a call-by-value parameter and a call-by-reference parameter. For the latter, if it belongs to a function parameter-list, an error message (FUNCTIONS CANNOT HAVE VAR PARAMETERS) will be printed out. Next corresponding to the subprogram identifier will point to the first parameter identifier, if there is any, otherwise it will be nil. Next corresponding to each parameter identifier will point to the next parameter identifier in the parameter-list, otherwise it will be nil. This procedure will start by processing the left-parenthesis and will terminate by processing the right parenthesis.

After the last sub-program declaration will have been processed by the procedure PROCDECLARATION, the procedure DECLARATIONPART will terminate and that will cause a call to the procedure BODYPART from within the main program.

3.3.3 Linking Names with Actions

BODYPART, which is the third procedure to be called from within the main program body, builds a linked chain of records of type PARSER. The first record of the chain, corresponding to the first statement of the body, is pointed to by Bodi belonging to its sub-program or program identifier, which is at the top of the stack Display.

Each statement record of type PARSER has the following fields:

Nextr : Pointer to a record of type PARSER, containing the next statement ;

Variant fields :

Assign, Proced : Stores an assignment or a procedure call statement. Identi points to the identifier being assigned in the first case, or to the identifier of the procedure being called in the second.

Expl points to the expression which is assigned to the identifier pointed by Identi in the first case, or to a list of expressions corresponding each to an actual value to be given to each of the formal parameters in the second case ;

Compound : Indicates the beginning of a compound statement. Ctr0 points to the first compounded statement ;

Condstate: Stores an if-then-else statement.
Exp2 points to the boolean expression of the conditional statement.
Ctr1 points to the statement-list following the 'then' symbol.

Ctr 2 points to the statement-list following the 'else' symbol, if any, otherwise it is nil ;

Repstate: Stores a while-do statement.
Exp3 points to the boolean expression of the repetitive statement.
Ctr3 points to the statement-list following the 'do' symbol.

Expressions are kept in linked records of type EXPRES with the following fields :

Nexte : Pointer to a record of type EXPRES, containing the next token of the current expression ;

Variant fields :

Cst : Keeps the unsigned integer number ;

Iden : For a variable or function identifier, idenp points to its record of type IDENTIFIER ;

Oper : Keeps the operators in Ope ;

Subsymb: Keeps 'comma', 'left parenthesis', 'right parenthesis' and 'not' symbols in Sym .

BODYPART calls the procedure STATEMENT until the end of a body is encountered. Bodi field of the record corresponding to the sub-program or program identifier will point to the first statement of its body. Once the whole body is processed, the stack Display will be popped. If the stack is not yet empty, a new symbol will be read, and if it is not a 'begin' symbol, then the procedure BODYPART will terminate and the main program will call the procedure DECLARATIONPART again ; if it is a 'begin' symbol then the procedure BODYPART will keep on calling the procedure STATEMENT until the end of this new body. At the end

of a body, after having popped the stack Display, if the stack becomes empty, that will mean that the main program body has just been processed and the processing has reached the end of the input text. In this case the procedure BODYPART will terminate and the main program will call the procedure QUERY to ask if the whole program or only a part of it has to be translated into the FDL.

STATEMENT will create a record of type PARSER, and if the last symbol read was an identifier, will call the procedure SEARCHID to determine the pointer to its record in the tree. If it is a procedure identifier, the tag-field St of the record which has just been created will be set to proced and the procedure CALL will be called to store the actual values to its parameters into a chain of records of type EXPRES. If the identifier was a variable or a function identifier, then the tag-field St will be set to assign and the procedure ASSIGNMENT will be called to store the expression into a chain of records of type EXPRES. If the last symbol read was a 'begin' symbol, then St will be set to compound and COMPOUNDSTATEMENT will be called, if it was an 'if' symbol, then St will be set to condstate and IFSTATEMENT will be called, if it was a 'while' symbol, then St will be set to repstate and WHILESTATEMENT will be called.

CALL will create a chain of records of type EXPRES to store the actual values being assigned to the parameters. The first element of this chain of records will contain the left parenthesis by setting its tag-field kind to Lp, and the last element will contain the right parenthesis by setting its tag-field kind to Rp. In between, actual values separated by commas will be stored, by calling the procedure EXPRESSION, and a type check will be made for each actual value.

ASSIGNMENT will call the procedure EXPRESSION to store the expression into a chain of records of type EXPRES. The first record of the chain will be pointed to by Expl of the Assign field. A type check between the identifier and the expression will be made.

COMPOUNDSTATEMENT will call the procedure STATEMENT until the end of the compounded statement-list. A chain of records of type PARSER will be created for the whole compounded statement-list. The first record of the chain will be pointed to by Ctr0 of the Compound field.

IFSTATEMENT will call the procedure EXPRESSION to create a chain of records of type EXPRES to store its boolean expression. The first record of the chain will be pointed to by Exp2 of the Condstate field. A type check will be made to see if the type of the expression is boolean. The procedure STATEMENT will be called to create a record for the statement following the 'then' symbol, and Ctr1 of the Condstate field will point to that record of type PARSER. If there is an 'else' clause, then the procedure STATEMENT will be called again to store the statement following the 'else' symbol and Ctr2 will point to that record, otherwise Ctr2 will be nil.

WHILESTATEMENT will call the procedure EXPRESSION to store its boolean expression. Exp3 of the Repstate field will point to the first record of that expression which must be of type boolean. After the type check, the procedure STATEMENT will be called to create a record of type PARSER for the statement following the 'do' symbol, and Ctr3 of the Repstate field will point to that record.

EXPRESSION will call the procedure SIMPLEEXPRESSION, after which, if there is a relational operator, it will be stored in a record of type EXPRES, and that new record will be appended to the chain of records of the same type created by SIMPLEEXPRESSION ; then the procedure SIMPLEEXPRESSION will be called again to store the second half of the expression which will be appended at the end of the chain. A type check will be made to see if the comparison is possible between these two simple expressions.

SIMPLEEXPRESSION will store the sign in a record of

type EXPRES, if there is a minus sign, then will call the procedure TERM. If signed and if the type of the term is not 'integer' then an error message will be printed out. If an 'addition', 'substraction' or 'or' operator follows then it will be stored in a record of type EXPRES and appended at the end of the chain created by TERM and the procedure TERM will be called again to store the second operand, after which a check will be made to see if both operands are legal to the operator. Then the whole chain will become the first operand if another 'additive' operator follows, and the same will be repeated again until no 'additive' operator is left.

TERM will start by calling the procedure FACTOR. After that, as long as there is a 'multiplicative' operator (e.g. 'multiplication' 'integer division', 'modulo operation' or 'and' operation), it will be stored, the procedure TERM will be called and the two operands will be compared to see if they are both legal to the operator. Then the whole chain will become the first operand and the same will be repeated again until no 'multiplicative' operator is left.

FACTOR will create a record of type EXPRES and depending on the last symbol read will take different actions. If the last symbol read was an identifier, then the tag-field will be set to Iden, the procedure SEARCHID will be called to get the pointer to its record in the tree, and this pointer value will be assigned to idenp of the field iden, if it is a function identifier, then the procedure CALL will be called to store the actual values to its parameters in a chain of records of type EXPRES which will be appended to the first record created by FACTOR. If the last symbol was an unsigned integer, the tag-field kind will be set to Cst and the integer be stored in Cval. If the last symbol was a left parenthesis, then the tag-field kind will be set to Subsymb and the symbol be stored in Sym. The procedure EXPRESSION will be called to store the expression which follows in a chain of records of type EXPRES which will be appended to the record created by FACTOR. A record containing the closing right parenthesis will be appended to the end. If the last symbol read was a 'not' symbol, it will be stored and the

procedure FACTOR will be called to deal with the rest. After which a type check will be made to see that the last factor which was analysed was of type boolean.

That ends the description of the procedure BODYPART, and also of the whole data structures. The last procedure QUERY, which will be called by the main program, will use the data structures created so far, but will not create new data structures. It will translate them into a FDL text which can then be processed by the Analyzer.

3.3.4 Translation into FDL

Once the main program body of the input text has been processed by BODYPART, the main program calls QUERY to ask the user if the program or a sub-program of the input text is to be analysed. The user types in the identifier of the program or a sub-program. Only the first eight characters of this identifier are read, and if the identifier has less than eight characters, a 'blank' character must be typed in after the last character. QUERY then looks into the array DICT to find if the identifier is in the dictionary. If the identifier is pointed to from one of the DICT entries, then its translation is performed by calling the procedure FLOWCHART. If the identifier is not in DICT, then a message (NAME NOT FOUND IN DICT) is printed out, and the user is asked again if another program or sub-program is to be analysed. This process is repeated until the user answers 'THATSALL'.

FLOWCHART prints out 'TITLE' PROGRAM/PROCEDURE/ FUNCTION (one of these) ⟨identifier⟩ ; and then declares standard type 'character' and standard infix operators 'modulo' and 'integer division', by printing out :

'TYPE' CHAR ;
'INFIX' MOD (INTEGER, INTEGER):INTEGER ;
'INFIX' DIV(INTEGER, IN TEGER):INTEGER;

The procedure DECLARE is then called, to print out the declarations of all the identifiers which are visible at the level requested by the user. After the control returns from the procedure DECLARE, the line 'START' will be printed out and the procedure TRANSLATE will be called to translate into the FDL the body of the program or sub-program asked by the user. After the return of control from the procedure TRANSLATE, the line 'FINISH' will be printed out to indicate the end of the body, and at this point the procedure FLOWCHART terminates. The fact that the procedures DECLARATIONPART and BODYPART were independent one from the other, each with its own data structures, has enabled us to have two independent procedures DECLARE and TRANSLATE to deal with the translation into the FDL of the declarations, and program or sub-program bodies respectively.

DECLARE needs to find the scope corresponding to the program or sub-program to be analysed. If the main program is to be analysed, all the identifiers can be used by successive calls to the nested sub-programs, therefore they all must be declared. The variable and type declarations at the main program level are both visible inside the whole program, making it necessary to translate them in any case. But we have to make a distinction between the call-by-value parameters and the local variables of the sub-programs in one hand, and the call-by-reference parameters of the sub-programs in the other. If the main-program is to be analysed, all the former must be translated after having renamed them to avoid name-clashes ; the latter need not be translated, as each time a sub-program call is made, only one of the former or a main program variable can be passed as a call-by-reference parameter. We can illustrate these points with the following example :

```
PROGRAM ALPHA(INPUT, OUTPUT);
 TYPE ⟨type declarations⟩;
 VAR A:INTEGER;B:BOOLEAN ; C:CHAR ;
 ⟨Subprograms⟩ ;
```

```
PROCEDURE BETA(A:INTEGER;VAR D:BOOLEAN);
   VAR K:INTEGER ;DELTA:BOOLEAN ;
   〈Subprograms〉;
   PROCEDURE THETA(L:INTEGER ; VAR DUMMY : BOOLEAN) ;
   BEGIN
   〈statement5〉
   END ;
 BEGIN(*BETA*)
 〈statement3〉;
  THETA (K+2*A, D) ;
 〈statement4〉 ;
  THETA(A, DELTA)
 END(*BETA*);
 〈sub-programs〉 ;
 BEGIN(*ALPHA*)
 〈statement1〉;
  BETA(A+2*ORD(C), B);
 〈statement2〉
 END(*ALPHA*).
```

If the user asks for ALPHA to be analysed, all the variable and type declarations at the level of ALPHA must be translated. As we want to unwrap the sub-programs at the point of their call, their local variables and call-by-value parameters need also be translated, as only their body will use these identifiers, their value at the end of the body being of no interest. On the other hand, call-by-reference parameters will be able to change the value of the parameters or local variables of the other sub-programs or even the variables of the main program. Therefore, if we declare and assign to them the actual parameter at the point of the call, as we do for the call-by-value parameters, we will have to re-assign the value of each call-by-reference parameter back to the corresponding actual parameter, at the end of the called sub-program. Leaving aside for the moment the issue of renaming, we can illustrate this point as follows from the PROGRAM ALPHA example :

The call THETA(A##1, DELTA ##1) in the procedure BETA would give

```
L##2:=A##1 ;
DUMMY##2:=DELTA##1 ;
⟨body of THETA⟩ ;
DELTA ##1:=DUMMY##2 ;
```

Two assignments and one declaration can be saved, for each call-by-reference parameter, if we decide to use DELTA##1 in the body of THETA, which would give :

```
L##2:=A##1 ;
⟨body of THETA using DELTA##1⟩ ;
```

We have adopted this solution, avoiding the unnecessary declarations and assignments which would have 'littered' the translated FDL text.

Leaving aside the declaration of standard 'character' type and 'mod', 'integer division' infix operators, we can give the translation of ALPHA into the FDL :

```
'TITLE'PROGRAM ALPHA;
'TYPE' ⟨type declarations⟩;
'VARIABLE' A:INTEGER ;
'VARIABLE' B:BOOLEAN ;
'VARIABLE' C:CHAR ;
'VARIABLE' A##1:INTEGER ;    {call-by-value parameter A of BETA}
'VARIABLE' K##1:INTEGER ;
'VARIABLE' DELTA##1: BOOLEAN ;  {local variables of BETA}
'VARIABLE'L##2:INTEGER ;    {call-by-value parameter L of THETA}
.....    {declarations of variables of other sub-programs
          nested in ALPHA}
'START'
 ⟨statement1⟩;
A##1:=A+2*ORD(C) ;    {translation of BETA(A+2*ORD(C), B)starts}
```

```
⟨statement3⟩;          {uses B instead of its call-by-reference parameter D}
L##2:=K##1+2*A##1 ;    {translation of THETA(K+2*A, D) starts here}
⟨statement5⟩;          {uses B instead of its call-by-reference parameter
                                                          DUMMY}
⟨statement4⟩ ;         {uses B instead of its call-by-reference parameter D}
L##2:=A##1 ;           {translation of THETA(A, DELTA)starts here}
⟨statement5⟩;          {uses DELTA##1 instead of its call-by-reference
                         parameter DUMMY}
⟨statement2⟩;
'FINISH'
```

We still have not explained yet how to deal with the function calls inside the expressions. Obviously, the same considerations as for the procedure calls apply to them, with the simplification that functions cannot have call-by-reference parameters in Pascal-Minus, avoiding the side-effects inside them. But, there is one major complication which is that several calls to the same function can be made in a single expression. So, if we want to expand the function body for each instance of these calls, at each instantiation, a different name must be given to the function identifier, in order to enable us to use them later in the expression. This problem is solved, by counting the maximal number of times a function identifier can appear in a single expression, and by declaring them as variables with a '### ⟨digit⟩' ending. In this way a maximal number of 10 instantiations of the same function in a single expression can be dealt with. The expression

```
A:=ADD(B, ADD(C, D))*100;
```

would be translated as :

```
⟨function body ADD with actual parameters C, D and the result
being assigned to ADD###0⟩;
⟨function body ADD with actual parameters B, ADD###0 and the
result being assigned to ADD # #1⟩ ;
A:=ADD# #1*100 ;
```

That ends the explanation of the functioning of DECLARE for the case

the main program of the input Pascal-Minus text is to be translated.

3.3.5 Translating Sub-Routines Separately

If the user asks for a sub-program to be translated, we have to find the scope at that level. As an example, if the sub-program to be translated is BETA, not only we have to declare the type and variable definitions of the main program ALPHA and, all the call-by-value parameters and local variables of all the sub-programs between the heading of ALPHA and BETA, we also have to declare all the call-by-value parameters and local variables starting from the heading of BETA and ending when we meet the body of BETA. To this end DECLARE calls the procedure TRAVERSE to determine this depth, after which all the declarations between the main program heading and the depth are translated except for the call-by-reference parameters.

As for the call-by-reference parameters, although none of them need be declared if the main program is to be analysed, in the case of a sub-program, its own call-by-reference parameters, as well as those of other sub-programs which contain the sub-program to be analysed, must be declared after having been renamed. As an example, we can take the case of THETA being asked for analysis. That would give the following translation into the FDL:

'TITLE' PROCEDURE THETA ;
'TYPE' ⟨type declarations of ALPHA⟩;
'VARIABLE' A:INTEGER ;
'VARIABLE' B:BOOLEAN ;
'VARIABLE' C:CHAR;
'VARIABLE' A#1:INTEGER ;
'VARIABLE' D#1:BOOLEAN ;
'VARIABLE'K#1 :INTEGER :
'VARIABLE' DELTA #1:BOOLEAN ;
'VARIABLE' L#2:INTEGER;
'VARIABLE' DUMMY#2:BOOLEAN;
'START'
⟨statement 5⟩; {this statement could use any of the declared variables}
'FINISH'

This is exactly how the procedure DECLARE proceeds. After having found the depth, by calling the procedure TRAVERSE, declared all the variable and type definitions of the main program and, all the call-by-value parameters and local variables of the sub-programs that can be reached from the level of analysis by calling the procedure ACTVARS, it will also declare the function identifiers from within ACTVARS by calling the procedure COUNTEXP to count the maximal number of occurrences of a function identifier in a single expression for a given depth, as explained above.

After which, depending on the level of analysis, the call-by-reference parameters of the sub-program being analysed will be declared by calling the procedure FORMVARS. As explained above, if the sub-program being analysed is nested in other sub-programs, their call-by-reference parameters must be declared as well. This is checked by calling the procedure ISIN and the necessary call-by-reference parameters are declared by calling the procedure FORMVARS again. That ends the description of the functioning of the procedure DECLARE.

3.3.6 Translation of Each Statement

The last stage consists of translating the body of the program or sub-program into the FDL. This is done by calling TRANSLATE from within FLOWCHART.

As it has already been explained in the description of BODYPART, Pascal-Minus possesses five kinds of statement, namely : assignment, procedure call, compound, conditional (if-then-else) and repetitive (while-do). We have to describe now, how each of these is going to be translated into the FDL, by traversing the tree constructed by BODYPART.

For an assignment statement, the procedure CHECKEXP will be called to see if there is a function call in the expression. If the procedure CHECKEXP finds a function call, it will then call the

procedure PFCALL to assign the actual values to the formal parameters of the function. At that point, the procedure CHECK1 will be called to see if the actual values contain any function calls, and if there is any, to translate them by calling the procedure PFCALL from within the procedure CHECK1. After this check, the actual values to the call-by-value parameters will be assigned by printing out an assignment statement in the FDL text ,and for the call-by-reference parameters, they will be renamed after the name of the actual parameter being passed in the call. Once the parameter-passing mechanism used in a compiler has been 'mimicked' for all the parameters, the procedure PFCALL will call the procedure TRANSLATE to translate the function body into the FDL and the result-value will be assigned to an identifier of kind :

⟨function identifier truncated to eight characters⟩ # # ⟨digit⟩

as explained above. It is that identifier which will be used in the expression in place of the function call. Once all the function calls will have been translated by the help of the procedure PFCALL, the original assignment will be copied out with its function instantiations without changing the structure of the expression. This is due to the fact that Pascal-Minus and FDL both have the same syntax for assignments and expressions.

For a procedure call, the procedure PFCALL will be called to 'mimic' the parameter-passing mechanism and to 'unwrap' its body. If this procedure call is the last statement in a statement-list, to avoid putting more than one label at the end of the translation of the procedure-call, the label which would have been put at the end of the statement-list, will be passed to the procedure PFCALL. This label is in Endlab and there is a flag attached to it (Towrite) to determine if it is to be written. If at the end of the procedure body, this label is used and printed out, then Towrite will be set to 'false' to avoid printing it again after returning from the procedure PFCALL. The same mechanism will be used for statements possessing statement-lists as their part.

For a compound statement, all that has to be done is to pass the Ctr0 pointing to the first statement in the list, together with Endlab, to the procedure TRANSLATE again.

For a conditional statement, its boolean expression must be checked by calling the procedure CHECKEXP, in the same way as for the expression of the assignment statement. After which, the following translation is made :

IF ⟨condition⟩ THEN ⟨statement1⟩ ;
will become :
'IF' NOT (⟨condition⟩)
'THEN' 'GOTO' α ;
⟨statement 1⟩;
α :

In the same way,
IF ⟨condition⟩ THEN ⟨statement1⟩ ELSE ⟨statement 2⟩ ;
will become :
'IF' NOT (⟨condition⟩)
'THEN' 'GOTO' α ;
⟨statement 1⟩ ;
'GOTO' β ;
α: ⟨statement 2⟩;
β :

To control the production and the printing of the labels, two procedures ENDLABEL and PRINTLAB are used. In this way, only the strict minimal number of labels are produced and each label is printed (as one expects it) only once.

For a repetitive (while-do) statement, we have to start by labelling the current line, if it has not already been made. The boolean variable Labeled keeps record of this fact together with Lbl which has the value of the last printed label. After that, its boolean expression must be checked by calling the procedure CHECKEXP,

in the same way as for the assignment or conditional statements.

WHILE ⟨condition⟩ DO ⟨statement⟩ ;
will be translated as :

```
α:'IF' NOT ( ⟨condition⟩ )
    'THEN' 'GOTO' β ;
   ⟨statement⟩ ;
    'GOTO' α ;
β :
```

If there is any function call in the ⟨condition⟩, this must be evaluated each time the loop will be executed. To that effect, α is printed first, and the procedure CHECKEXP called to 'unwrap' the function calls inside the ⟨expression⟩ starting from the label α. The following example illustrates this point :

WHILE ADD(A, B) > 0 DO ⟨statement⟩ ;
will be translated as :

```
α : ⟨function body ADD with actual parameters A, B and the
      result being assigned to ADD # # 0⟩;
 'IF' NOT (ADD # # 0 > 0)
 'THEN' 'GOTO' β ;
 ⟨statement⟩;
 'GOTO' α ;
β :
```

After all the statements belonging to the body of the program or sub-program will be translated, the control will return to QUERY again as explained above.

3.4 Extensions :

As we had already anticipated, the Pascal-Minus language and its translator into the FDL, the Program PMINUS are open to extensions.

For practical reasons, one may want to introduce other

control structures from Pascal, or, new and more powerful constructs can be added. We will give hereafter a few examples.

The introduction of repeat-until and for-do loops can easily be done without necessitating big changes to the data or program structure of PMINUS. The statement

REPEAT ⟨statement⟩ UNTIL ⟨expression⟩ ;

could be translated as :

α : ⟨statement⟩ ;
'IF' NOT ⟨expression⟩
'THEN' 'GOTO' α ;

Or the statement FOR V:=E1 TO E2 DO ⟨statement⟩ ;
could be translated as :

V:=E1;
α: 'IF' V > E2 'THEN' 'GOTO' β ;
⟨statement⟩;
V:=SUCC(V);
'GOTO' α;
β:

The statement FOR V:=E1 DOWNTO E2 DO ⟨statement⟩ ;
could be translated as :

V:=E1;
α : 'IF' V < E2 'THEN' 'GOTO' β;
⟨statement⟩;
V:=PRED(V);
'GOTO' α ;
β :

To introduce these two constructs, the only change to the data structures should be made to the records of type PARSER. These would become :

```
STATEMENTKIND=(ASSIGN, PROCED, COMPOUND, CONDSTATE,
                    WHILELOOP, REPEATLOOP, FOR LOOP);

PARSER=RECORD
          NEXTR:CTR;
          CASE ST:STATEMENTKIND OF
           ASSIGN, PROCED:(IDENTI:CTP;EXP1:EXP);
           COMPOUND:(CTRO:CTR);
           CONDSTATE :(EXP2:EXP;CTR1, CTR2:CTR);
           WHILELOOP, REPEATLOOP:(EXP3:EXP;CTR3:CTR) ;
           FORLOOP:(UP:BOOLEAN;EXP4, EXP5:EXP;CTR4:CTR)
        END ;
```

Obviously for a repeat-loop , as for a while-loop , Exp2 will point to the boolean expression, and Ctr3 to the first statement of the statement-list contained in the loop.

For a for-loop, up will be 'true' if the index is to be incremented at each execution of the loop, 'false' otherwise. Exp4 will point to E1 and Exp5 to E2 of the above example. Ctr4 will point to the first statement of the statement list contained in the loop. Two procedures REPEATLOOPB and FORLOOPB must be written to be inserted into STATEMENT to deal with these loop constructs, just as it was done for while-do loop construct.

It is also possible to replace all these loop constructs with just one general loop construct with multiple exits [DAN82]. This would give :

```
loop
   ⟨statement-list1⟩
endloop
   ⟨select-statement⟩;
```

One or more statements in the ⟨statement-list 1⟩ could have the form :

if ⟨expression I⟩ then exit ⟨label I⟩;

The ⟨select-statement⟩ would consist of :

```
select
   ⟨label1⟩ ⇒ ⟨statement1⟩ ;
      :          :
   ⟨labelI⟩ ⇒ ⟨statementI⟩ ;
    other   ⇒ ⟨statement-other⟩
end select ;
```

Any of the statements at the right of '⇒' sign can be an exit statement as the loop statements may be nested.

If this approach were adopted, the while-do, repeat-until and for-do loops could be replaced as follows :

WHILE ⟨expression⟩ DO ⟨statement1⟩ ; ⟨statement2⟩ ;
would become :

```
LOOP
  IF NOT ⟨expression⟩ THEN EXIT LAB1 ;
 ⟨statement1⟩
ENDLOOP;
SELECT
   LAB1 ⇒ ⟨statement2⟩
ENDSELECT ;
```

REPEAT ⟨statement 1⟩ UNTIL ⟨expression⟩ ; ⟨statement2⟩ ;
would become :

```
LOOP
  ⟨statement1⟩ ;
  IF ⟨expression⟩ THEN EXIT LAB1
ENDLOOP;
SELECT
   LAB1 ⇒ ⟨statement2⟩
ENDSELECT ;
```

FOR V:=E1 TO (respectively DOWNTO) E2 DO ⟨statement⟩ ;
⟨statement2⟩

would become:

```
V:=E1 ;
LOOP
  IF V > (respectively <) E2 THEN EXIT LAB1 ;
 ⟨statement1⟩;
V:=SUCC (respectively PRED)(V) ;
ENDLOOP;
SELECT
  LAB1 ⇒ ⟨statement 2⟩
ENDSELECT ;
```

The general loop construct could be translated into the Flowchart Language as :

```
α: ⟨statements⟩ ;
'IF' ⟨exp1⟩ 'THEN' 'GOTO' LAB1 ;
  :
'IF' ⟨expN⟩'THEN' 'GOTO' LABN ;
⟨other statements⟩;
'GOTO'α;
LAB1: ⟨statement1⟩;
'GOTO' β ;
  :
LABN: ⟨statementN⟩ ;
β:
```

There is a strong case against the use of the 'GOTO' statement, especially from the verification point of view, but systems dealing with asynchronous events and interrupts may still need it, therefore it can be added into the Pascal-Minus and the users asked to be very careful about the choice to insert it in their programs. As the 'GOTO' statement exists in the FDL , its translation would be straightforward. The records of type PARSER, dealing with the statements could have an additional

statementkind, with a field to store the label to which a jump must be made. Each statement would also need a label field, thus PARSER becoming :

```
PARSER=RECORD
          NEXTR:CTR;LB:INTEGER;
          CASE ST: STATEMENTKIND OF

           GOTOSTATE :(LBJ:INTEGER)
       END;
```

with the STATEMENTKIND=(ASSIGN, .. , GOTOSTATE);

LB would contain the label of the statement being stored in this actual record, and LBJ would contain the label of the statement to which a jump must be made. A procedure GOTOSTATEMENT would be added in STATEMENT.

4. VERIFYING A PASCAL-MINUS PROGRAM

To illustrate different points made in the preceding chapters, a program in Pascal-Minus was written, and translated into the FDL by using the program PMINUS described in the previous chapter. The program BURNERCONTROLLER and its translated form can be found in Appendix II.

The program written in Pascal-Minus gives a high-level description of a burner-controller, together with assertions in the form of Boolean expressions which must hold whenever the execution reaches them.

The example system controls the timing and the re-ignition of a gas-burner. The user enters through a keypad, the period during which heating is required (on-time $\leq$ heating-period $\leq$ off-time). The user can also turn the heating on or off outside the required heating-period by pressing manual-on/manual-off keys.

Once the system decides that the burner should be on, it goes through an ignition phase. If successful, it remains in this state until the end of the required heating-period, otherwise it locks out all further attempts at re-ignition from timed and manually setting-on, until the 'reset lock-out' switch has been depressed. If after a successful ignition phase, the flame goes out, then a new ignition phase starts to keep it on during the whole heating-period.

Only five successive re-ignition attempts can be made, by turning on both the burner-valve and the spark ignition relays. If a flame is detected, then the spark-drive is turned-off, leaving the burner-relay on, otherwise both relays will be turned off before another attempt is made.

The system needs the time of day to decide for the

required heating-period. The function REALTIME returns it, for being assigned to the variable real. NEWKEY and FLAME are also interface functions, returning the information about the existence of a new entry from the keypad, and, about the flame being on or off, respectively. The procedure IO is called to deal with the entries to the system from the keypad.

As the implementation is expected to be done on a microprocessor, regular checks are made to assure that, first of all, the area of the memory reserved for the actual program has not been overwritten, and that the sum of time variables (i.e. real, on and off) equals an updated checksum. These checks are done by calling the functions PROGMEM and UPDATED respectively. If the check on the program memory fails, a PROGRAM-LED is turned on ; if the sum of real, on and off does not equal the updated checksum, a DATA-LED is turned on, blocking the system until a new power-on-reset.

Similarly, five successive ignition failures will cause an IGNITION-LED which can only be eliminated by a reset-lockout.

If none of these faults occur, the assertion

{ NOT(PROGRAM-LED OR DATA-LED OR IGNITION-LED) }

will hold. In that case, the function REQUIRED will be called to decide if heating is required. If it is required and the gas active with flame on, then no action is taken, otherwise the procedure IGNITE will be called for a new attempt at re-ignition. If five unsuccessful attempts had already been made, then an IGNITION-LED is turned on.

After the termination of the procedure IGNITE, the assertion

{ NOT(PROGRAM-LED OR DATA-LED) AND
(IGNITION-LED AND ATTEMPT=5 and GAS=PASSIVE OR
NOT(IGNITION-LED)AND (FLAME AND ATTEMPT=0 OR
GAS=PASSIVE AND ATTEMPT $>$ 0)) }

will hold. It must be noticed that GAS=PASSIVE $\Rightarrow$ NOT FLAME,

as one could not expect to find the flame on without turning the burner-relay on (i.e. gas:=active). Whenever the flame is obtained, attempt will be put to 0, therefore the assertion $\{ \text{FLAME} \Longrightarrow \text{GAS=ACTIVE AND ATTEMPT=0} \}$ will always hold.

The program BURNERCONTROLLER is then input to the program PMINUS described in the previous chapter to obtain its translation into the FDL (See in Appendix II). The FDL text is first reduced by putting together program nodes which are not essential to the proof, keeping only the nodes at which we want a property of the system to hold. After that reduction is done by the existing facilities in the Department, the reduced form, (which can be seen in Appendix II) is symbolicly executed, as it was anticipated at the beginning of the previous chapter.

The symbolic execution is done step by step on the reduced text, starting from the node 1. Input values could have been separately assigned to the program variables at the beginning of the symbolic execution to simulate particular paths, but our implementation was already containing an initialization procedure to that purpose, and, the procedure IO which deals with the keyboard, together with the function REQUIRED, were not expanded in order to keep our interest focused on the main function (i.e. ignition when necessary) together with security checks. We will concentrate on the ignition phase to check, that an attempt at the re-ignition is made whenever heating is required ; that after an ignition failure the burner-relay is turned off (i.e. gas:=passive); that after five consecutive failures, the IGNITION-LED is turned on; that no further ignition attempts will be made before the reset-lockout; that while gas is active and heating required, if the flame goes off, a new ignition attempt will be made; and that whenever heating is no longer required, gas will be turned off.

These checks are made, by following the traversal conditions,

given by the symbolic execution, to find out that the assertion at that point in the original Pascal-Minus program will hold for the corresponding path function values.

As an example, we can see (Appendix III), at pathlength 4 for node 6, which corresponds to the line 70 in the original Pascal-Minus program, that if an ignition attempt fails, gas is turned off and the number of unsuccessful attempts incremented. The next traversal condition expresses the case in which the ignition-attempt was successful, keeping the gas on, and the number of unsuccessful attempts at 0.

These checks are made on all the executable paths given by the symbolic execution to make it sure that all assertions inserted into the Pascal-Minus program will hold whenever the execution reaches them. Some of these paths, together with the point reached by the execution in the Pascal-Minus program and the assertion attached to that point, are given in Appendix III.

Obviously, other specified Pascal-Minus programs must be checked in the way described above, in order to be able to evaluate fully its use and potentialities.

REFERENCES AND BIBLIOGRAPHY

For a more detailed and annotated bibliography, see pp. 243-271 in [YEH77] and pp. 269-319 in [YEH77a].

[ABR79] ABRIAL, J. R.; SCHUMAN, S. A.; MEYER, B.
'Specification Language' in [MCK80]

[AMB77] AMBLER, A. L. et al.
'GYPSY: A Language for Specification and Implementation of Verifiable Programs'
Proc. ACM Conf. on Language Design for Reliable Software, SIGPLAN Notices, pp 1-10, Mar. 1977.

[AMM81] AMMANN, U. 'The Zurich Implementation', in [BAR81]

[AND78] ANDERSON, E. R.; BELZ, F. C.; BLUM, E. K.
'Extending an Implementation Language to a Specification Language'
Lecture Notes in Computer Science, Vol. 75, pp. 385-424, 1979.

[BAN77] BANATRE, M.; COUVERT, A.; HERMAN, D.; RAYNAL, M.
'Types Abstraits et Objets Conservés'
IRISA-Rennes, Sept. 1977.

[BAN78] BANATRE, M.; COURVERT, A.; HERMAN, D,; RAYNAL, M.
'Types Abstraits et Pluralité de Leurs Représentations à l'Exécution'
IRISA-Rennes, Feb. 1978.

[BAN81] BANATRE, M.; COUVERT, A.; HERMAN, D.; RAYNAL, M.
'An Experience in Implementing Abstract Data Types'
Software-Practice and Experience, Vol. 11, pp. 315-320, 1981.

[BAR81] BARRON, D. W. (Ed.)
'Pascal - The Language and its Implementation'
John Wiley and Sons Ltd., 1981.

[BEN80] BENTLEY, J. L.; SHAW, M.
'An Alphard Specification of a Correct and Efficient Transformation on Data Structures'
IEEE Trans. on Software Eng., Vol. SE-6, pp. 572-584, Nov. 1980.

[BER81] BERNSTEIN, A. J.; ENSOR, J. R.
'A Modula Based Language Supporting Hierarchical Development and Verification'
Software-Practice and Experience, Vol. 11, pp. 237-255, 1981.

[BERT79] BERT, D.
'La Programmation Générique-Construction de Logiciel, Spécification Algébrique et Vérification'
Thèse, Université Scientifique et Médicale de Grenoble, June 1979.

[BLI81] BLIKLE, A. J.
'On the Development of Correct Specified Programs'
IEEE Trans. on Software Eng., Vol. SE-7, pp. 519-527, Sept. 1981.

[BUR77] BURSTALL, R. M.; GOGUEN, J. A.
'Putting Theories Together to Make Specifications'
Proc. 5th Int. Joint Conf. on Artificial Int., pp. 1045-1058, Aug. 1977.

[CAR80] CARRÉ, B. A.
'Software Validation', in Advanced Techniques for Microprocessor Systems, edited by F. K. HANNA, Peter Peregrinus, 1980.

[CLI81] CLINT, M.
'On the Use of History Variables'
Acta Informatica, Vol. 16, pp. 15-30, 1981.

[DAH70] DAHL, O. J.; MYHRHAUG, B.; NYGAARD, K.
'The SIMULA 67 Common Base Language'
Norwegian Computing Centre, Oslo, Norway, 1968.

[DAN82] DANNENBERG, R. B.; ERNST, G. W.
'Formal Program Verification Using Symbolic Execution'
IEEE Trans. on Software Eng., Vol. SE-8, pp. 43-52,
Jan. 1982.

[DAR76] DARLINGTON, J.; BURSTALL, R. M.
'A System which Automatically Improves Programs'
Acta Informatica, Vol. 6, pp. 41-60, 1976.

[DAR80] DARLINGTON, J.
'The Synthesis of Implementations for Abstract Data Types'
Dept. of Comp. and Control, Imperial College, London,
CCD-80/4, Jan. 1980.

[DEM80] DEMUYNCK, M.; CHENUT, S.; NERSON, J. M.
'Système Zaide d'Aide à la Spécification'
EDF-GDF, France.

[DIJ68] DIJKSTRA, E. W.
'A Constructive Approach to the Problem of Program
Correctness'
BIT, Vol. 8, pp. 174-186, 1968.

[DIJ72] DIJKSTRA, E. W.
'Notes on Structured Programming' in 'Structured Programming'
Academic Press, N. Y. 1972.

[DIJ76] DIJKSTRA, E. W.
'A Discipline of Programming'
Prentice-Hall, 1976.

[ERN80] ERNST, G. W.; OGDEN, W. F.
'Specification of Abstract Data Types in MODULA'
ACM Trans. on Prog. Lang. and Systems, Vol. 2, pp. 522-543,
Oct. 1980.

[FLO67] FLOYD, R. W.
'Assigning Meanings to Programs'
Proc. Symp. in Applied Math., Vol. 19, pp. 19-32, 1967.

[FLON79] FLON, L.; MISRA, J.
'A Unified Approach to the Specification and Verification of Abstract Data Types'
Conf. on the Spec. of Reliable Software, Cambridge, April 1979.

[FRA78] FRANCEZ, N.; PNUELI, A.
'A Proof Method for Cyclic Programs'
Acta Informatica, Vol. 9, pp. 133-157, 1978.

[GAN81] GANNON, J.; McMULLIN, P.; HAMLET, R.
'Data-Abstraction Implementation, Specification and Testing'
ACM Trans. on Prog. Lang. and Systems, Vol. 3, pp. 211-223, July 1981.

[GER76] GERHART, S. L.; YELOWITZ, L.
'Observations of Fallibility in Applications of Modern Programming Methodologies'
IEEE Trans. on Software Eng., Vol. SE-2, pp. 195-207, Sept. 1976.

[GER79] GERHART, S. L.; WILE, D. S.
'Preliminary Report on the DELTA Experiment : Specification and Verification of a Multiple-User File Updating Module'
Conf. on the Spec. of Reliable Software, Cambridge, April 1979.

[GOG78] GOGUEN, J. A.
'Some Design Principles and Theory for OBJ-0, A Language to Express and Execute Algebraic Specifications of Programs'
Lecture Notes in Computer Science, Vol. 75, Springer Verlag, 1979.

[GRE77] GREIF, I.
'A Language for Formal Problem Specification'
Comm. ACM, Vol. 20, pp. 931-935, Dec. 1977.

[GUT75] GUTTAG, J. V.
'The Specification and Application to Programming of Abstract Data Types'
Ph. D. Thesis, Comp. Syst. Res. Group, Tech. Rep. CSRG-59, Dept. Comp. Sci., University of Toronto, 1975.

[GUT77] GUTTAG, J. V.
'Abstract Data Types and the Development of Data Structures'
Comm. ACM, Vol. 20, pp. 396-404, June 1977.

[GUT78] GUTTAG, J. V.; HORNING, J. J.
'The Algebraic Specification of Abstract Data Types'
Acta Informatica, Vol. 10, pp. 27-52, 1978.

[GUT78a] GUTTAG, J. V.; HOROWITZ, E.; MUSSER, D. R.
'Abstract Data Types and Software Validation'
Comm. ACM, Vol. 21, pp. 1048-1064, Dec. 1978.

[GUT80] GUTTAG, J. V.
'Notes on Type Abstraction (Version 2)'
IEEE Trans. on Software Eng., Vol. SE-6, pp. 13-23, Jan. 1980.

[GUT80a] GUTTAG, J. V.; HORNING, J. J.
'Formal Specification as a Design Tool'
Conf. Proc. of 7th ACM Symp. on Principles of Prog. Languages, Jan. 1980.

[HAB73] HABERMANN, A. N.
'Critical Comments on the Programming Language PASCAL'
Acta Informatica, Vol. 3, pp. 47-57, 1973.

[HOA69] HOARE, C.A.R.
'An Axiomatic Basis for Computer Programming'
Comm. ACM, Vol.12, pp. 576-583, Oct. 1969.

[HOA71] HOARE, C.A.R.
'Proof of a Program : FIND'
Comm. ACM, Vol.14, pp. 39-45, Jan.1971.

[HOA72] HOARE, C.A.R.
'Proof of Correctness of Data Representations'
Acta Informatica, Vol.1, pp. 271-281, 1972.

[HOA73] HOARE, C.A.R.; WIRTH, N.
'An Axiomatic Definition of the Programming Language PASCAL'
Acta Informatica, Vol. 2, pp.335-355,1973.

[HOL78] HOLT, R.C.; WORTMAN, D.B.; CORDY, J.R.; CROWE, D.R.
'The Euclid Language : A Progress Report'
Proc. of ACM National Conf., Washington, Dec. 1978.

[HOL80] HOLT, R.C.; WORTMAN, D.B.; CORDY, J.R.;
CROWE, D.R.; GRIGGS, I.H.
'The Toronto Euclid Compiler'
University of Toronto and I. P. Sharp Associates Ltd.,
Jan.1980.

[HOU80] HOUSE, R.
'Comments on Program Specification and Testing'
Comm. ACM, Vol. 23, pp. 324-331, June 1980.

[HUE80] HUET, G.
'A Complete Proof of Correctness of the Knuth-Bendix Completion Algorithm'
INRIA, Rapport de Recherche No. 25, July 1980.

[HUE80a] HUET, G.
'Proofs by Induction in Equational Theories with Constructors'
INRIA, Rapport de Recherche No. 28, Aug. 1980.

[HUE80b] HUET, G.
'Confluent Reductions : Abstract Properties and Applications to Term Rewriting Systems'
Journal of the ACM, Vol. 27, pp. 797-821, Oct. 1980.

[JEN75] JENSEN, K.; WIRTH, N.
'PASCAL-User Manual and Report'
Springer-Verlag, Berlin 1975.

[KNU70] KNUTH, D.E.; BENDIX, P.B.
'Simple Word Problems in Universal Algebras'
in 'Computational Problems in Abstract Algebra'
edited by J. LEECH, Pergamon Press, 1970.

[LAM77] LAMPSON, B.W. et al.
'Report on the Programming Language Euclid'
SIGPLAN Notices, Vol. 12, Feb. 1977.

[LEV78] LEVITT, K.N.; ROBINSON, L. ; HOROWITZ, E. ; LISKOV, B.
'Formal Methods in Programming - When Will They Be Practical?'
Proc. National Computer Conf., pp. 665-668, 1978.

[LIS75] LISKOV, B.H. ; ZILLES, S.N.
'Specification Techniques for Data Abstractions'
IEEE Trans. on Software Eng., Vol. SE-1, pp. 7-19, March 1975.

[LIS79] LISKOV, B.H.; SNYDER, A.
'Exception Handling in CLU'
IEEE Trans. on Software Eng., Vol. SE-5, pp. 546-558, Nov. 1979.

[LIS 80] LISKOV, B.H.
'Modular Program Construction Using Abstractions'
in 'Abstract Software Specifications', edited by D. Bjørner, Springer Verlag, pp. 354-389.

[LON78] LONDON, R.L. et al.
'Proof Rules for the Programming Language Euclid'
Acta Informatica, Vol. 10, pp. 1-26, 1978.

[MAJ77] MAJSTER, M.E.
'Limits of the "Algebraic" Specification of Abstract Data Types'
SIGPLAN Notices, Vol.12, pp. 37-42, Oct. 1977.

[MAJ79] MAJSTER, M.E.
'Data Types, Abstract Data Types and Their Specification Problem'
Theoretical Computer Science, Vol. 8, pp. 89-127, 1979.

[MCK80] McKEAG, R.M. ; MACNAGHTEN, A.M.
'On the Construction of Programs'
Cambridge University Press, 1980.

[MEY79] MEYER, B. ; DEMUYNCK, M.
'Specification Languages : A Critical Survey and Proposal'
EDF, France, Sept. 1979.

[MILL76] MILLEN, J.K.
'Security Kernel Validation in Practice'
Comm. ACM. Vol. 19, pp. 243-250, May 1976.

[MUS77] MUSSER, D.R.
"A Data Type Verification System Based on Rewrite Rules'
Proc. 6th Texas Conf. on Comp. Syst., Austin, Nov. 1977.

[MUS80] MUSSER, D.R.
'Abstract Data Type Specification in the AFFIRM System'
IEEE Trans. on Software Eng., Vol. SE-6, pp. 24-32, Jan. 1980.

[MUS 80a] MUSSER, D. R.
'On Proving Inductive Properties of Abstract Data Types'
Proc. 7th ACM Symp. on Principles of Prog. Lang.,
Jan. 1980.

[NOR81] NORI, K. V. et al.
'Pascal-P Implementation Notes' in [BAR81].

[PAR72] PARNAS, D. L.
'A Technique for Software Module Specification with Examples'
Comm. ACM, Vol. 15, pp. 330-336, May 1972.

[PAR72a] PARNAS, D. L.
'On the Criteria to be Used in Decomposing Systems into Modules'
Comm. ACM, Vol. 15, pp. 1053-1058, Dec. 1972.

[PAR75] PARNAS, D. L. ; SIEWIOREK, D. P.
'Use of the Concept of Transparency in the Design of Hierarchically Structured Systems',
Comm. ACM, Vol. 18, pp. 401-408, July 1975.

[PEM82] PEMBERTON
'Pascal Implementation'
Ellis Horwood, 1982.

[POP77] POPEK, G. J. et al.
'Notes on the Design of Euclid'
SIGPLAN Notices, Vol. 12, pp. 11-18, March 1977.

[PRY79] PRYWES, N. S. ; PNUELI, A. ; SHASTRY, S.
'Use of a Nonprocedural Specification Language and Associated Program Generator in Software Development'
ACM Trans. on Prog. Lang. and Systems, Vol. 1, pp. 196-217, Oct. 1979.

[REM76] REM, M.
'Associons and the Closure Statement'
MC Tract 76, Mathematical Centre, Amsterdam, Oct. 1976.

[REM81] REM. M.
'Associons : A Program Notation with Tuples Instead of Variables'
ACM Trans. on Prog. Lang. and Systems, Vol. 3, pp. 251-262, July 1981.

[ROB77] ROBINSON, L. ; LEVITT, K.N.
'Proof Techniques for Hierarchically Structured Programs'
Comm. ACM, Vol. 20, pp. 271-283, April 1977.

[ROB77a] ROBINSON, L.; LEVITT, K.N. ; NEUMANN, P.G.; SAXENA, A.R.
'A Formal Methodology for the Design of Operating System Software' in [YEH77].

[ROS73] ROSEN, B.K.
'Tree-Manipulating Systems and Church-Russer Theorems'
Journal of the ACM, Vol. 20, pp. 160-187, Jan. 1973.

[ROS77] ROSS, D.T.
'Guest Editorial Reflections on Requirements'
IEEE Trans. on Software Eng., Vol. SE-3, pp. 2-34, Jan. 1977.

[ROU77] ROUBINE, O. ; ROBINSON, L.
'SPECIAL Reference Manual'
Stanford Research Institute, Tech. Rep. CSL-45, Jan. 1977.

[SAY82] SAYI, H.
'From Abstract Specifications to Programs : A Distributed Approach'
Mini-Thesis, Dept. of Electronics, University of Southampton, March 1982.

[SHA79] SHAW, A. C.
'Software Specification Languages Based on Regular Expressions'
ETH, Institut für Informatik, Zürich, nr. 31, June 1979.

[SHA82] SHANKAR, K. S.
'A Functional Approach to Module Verification',
IEEE Trans. on Software Eng., Vol. SE-8, pp. 147-160, March 1982.

[SPI78] SPITZEN, J. M. ; LEVITT, K. N. ; ROBINSON, L.
'An Example of Hierarchical Design and Proof',
Comm. ACM. Vol. 21, pp. 1064-1075, Dec. 1978.

[SUF81] SUFRIN, B.
'Formal Specification of a Display Editor'
Oxford University, PRG-21, June 1981.

[SWA82] SWARTOUT, W. ; BALZER, R.
'On the Inevitable Intertwining of Specification and Implementation'
Comm. ACM, Vol. 25, pp. 438-440, July 1982.

[TAK80] TAKAHASHI, H.
'An Automatic-Controller Description Language'
IEEE Trans. on Software Eng., Vol. SE-6, pp. 53-64, Jan. 1980.

[TEN77] TENNENT, R. D.
'On a New Approach to Representation Independent Data Classes'
Acta Informatica, Vol. 8, pp. 315-324, 1977.

[WAL80] WALKER, B. J. ; KEMMERER, R. A. ; POPEK, G.
'Specification and Verification of the UCLA Unix Security Kernel'
Comm. ACM. Vol. 23, pp. 118-131, Feb. 1980.

[WEL77] WELSH, J. : SNEERINGER, W.J.; HOARE, C.A.R.
'Ambiguities and Insecurities in PASCAL'
Software-Practice and Experience, Vol. 7, pp. 685-696, 1977.

[WIL81] WILE, D.S.
'Type Transformations'
IEEE Trans. on Software Eng., Vol. SE-7, pp. 32-39, Jan. 1981.

[WIR71] WIRTH, N.
'Program Development by Stepwise Refinement'
Comm. ACM, Vol. 14, pp. 221-227, April 1971.

[WIR71a] WIRTH, N.
'The Programming Language PASCAL'
Acta Informatica, Vol.1, pp. 35-63, 1971.

[WIR71b] WIRTH, N.
'The Design of a PASCAL Compiler'
Software-Practice and Experience, Vol.1, pp. 309-333, 1971.

[WIR73] WIRTH, N.
'Systematic Programming'
Prentice-Hall, 1973.

[WIR74] WIRTH, N.
'On the Composition of Well-Structured Programs'
Computing Surveys, Vol. 6, pp. 247-259, Dec. 1974.

[WIR75] WIRTH, N.
'An Assessment of the Programming Language PASCAL'
IEEE Trans. on Software Eng., Vol. SE-1, pp. 192-198, June 1975.

[WIR81] WIRTH, N.
'Pascal-S : A Subset and its Implementation'
in BAR81 .

[WOR79] WORTMAN, D. B.
'On Legality Assertions in EUCLID'
IEEE Trans. on Software Eng., Vol. SE-5, pp. 359-367,
July 1979.

[WOR81] WORTMAN, D. B.; CORDY, J. R.
'Early Experiences with EUCLID'
Proc. Int. Conf. on Software Eng., San Diego, March 1981.

[WUL76] WULF, W. A.; LONDON, R. L.; SHAW, M.
'An Introduction to the Construction and Verification of
Alphard Programs'
IEEE Trans. on Software Eng., Vol. SE-2, pp. 253-265,
Dec. 1976.

[WUR81] WURGES, H.
'A Specification Technique Based on Predicate Transformers'
Acta Informatica, Vol. 15, pp. 425-445, 1981.

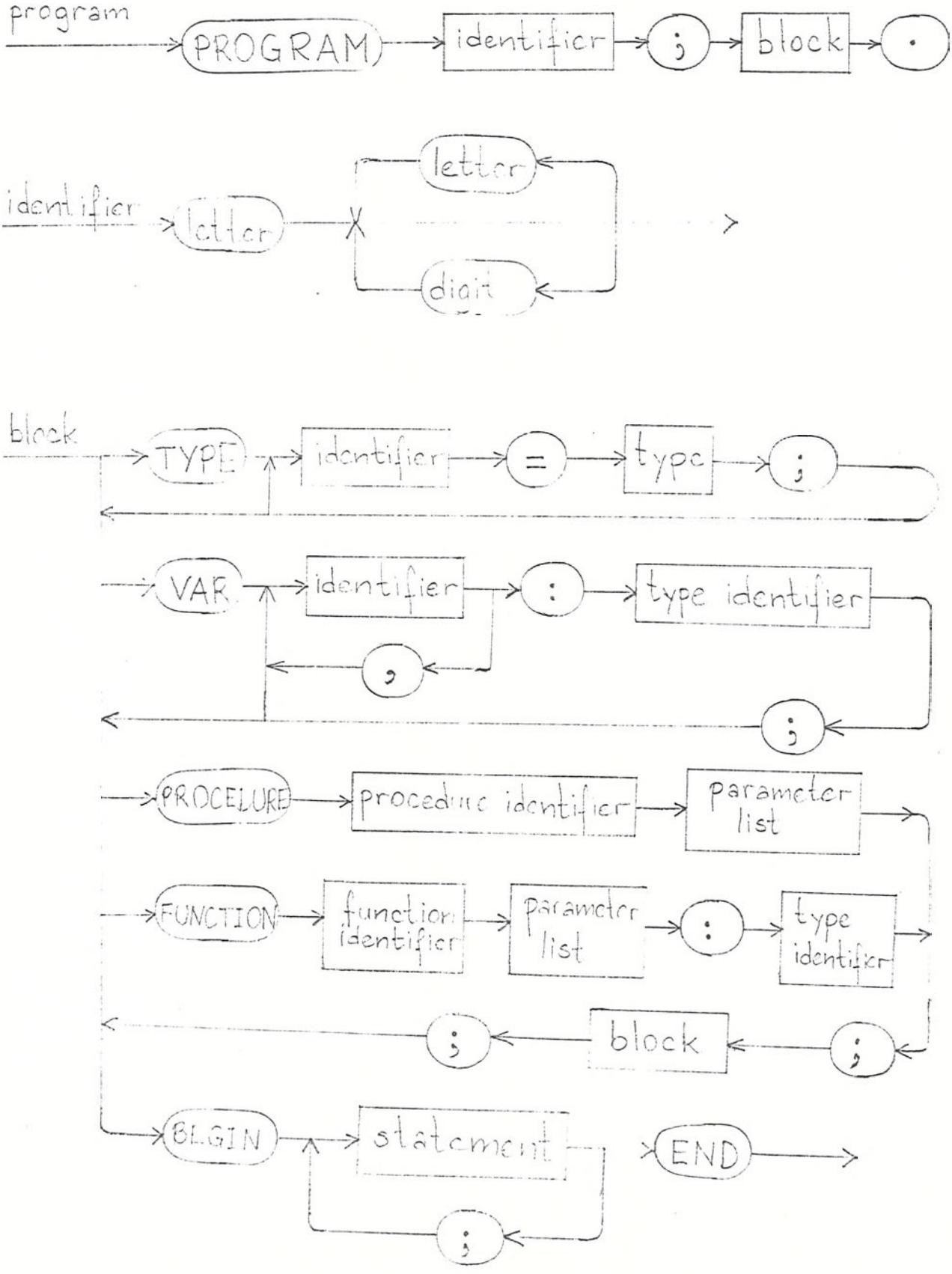
[YEH77] YEH, R. T.
'Current Trends in Programming Methodology'
Vol. 1, 'Software Specification and Design', 1977.

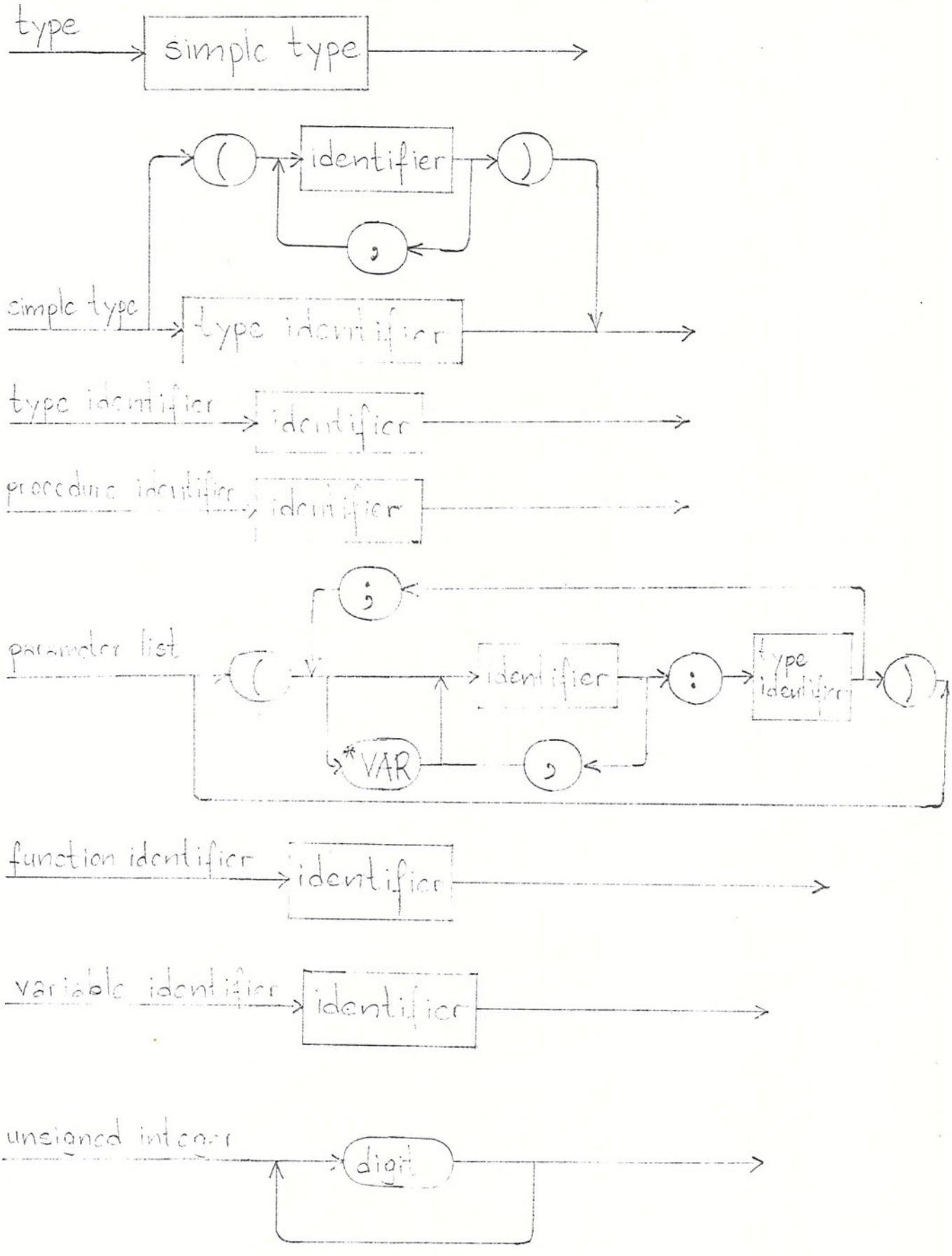
[YEH77a] Vol. 2, 'Program Validation', 1977.

[YEH78] Vol. 4, 'Data Structuring', 1978.
Prentice-Hall.

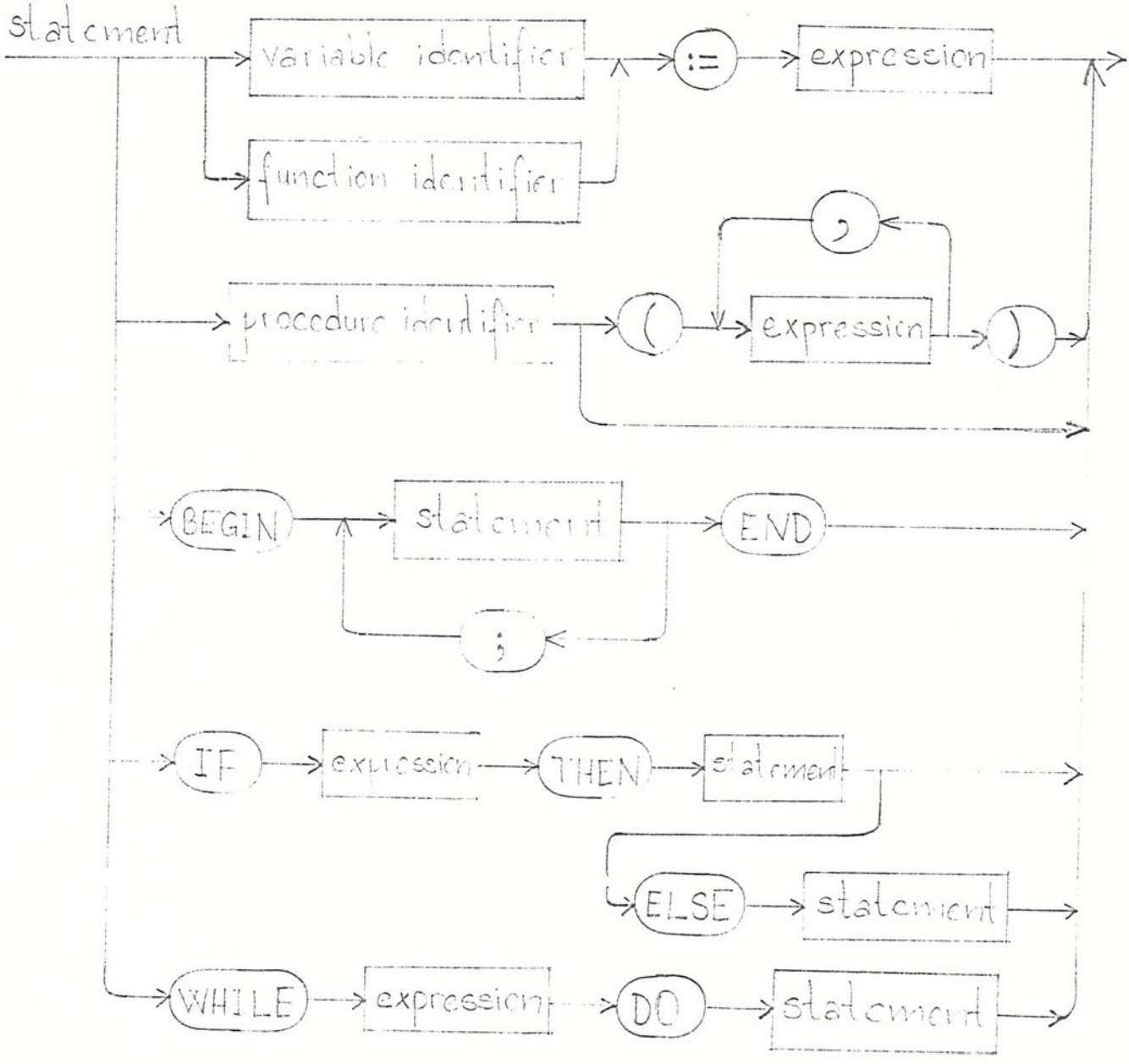
APPENDIX I

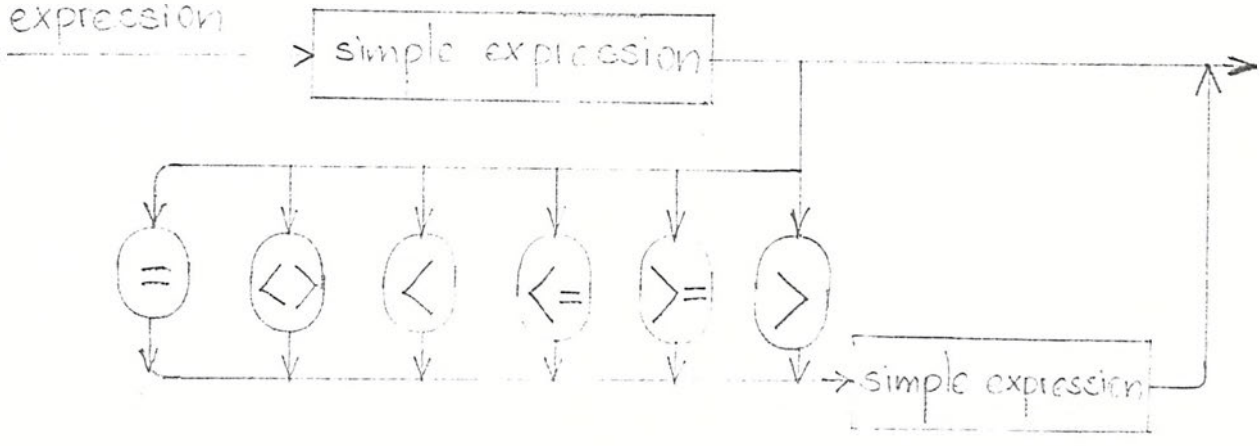
SYNTAX DIAGRAMS FOR PASCAL-MINUS

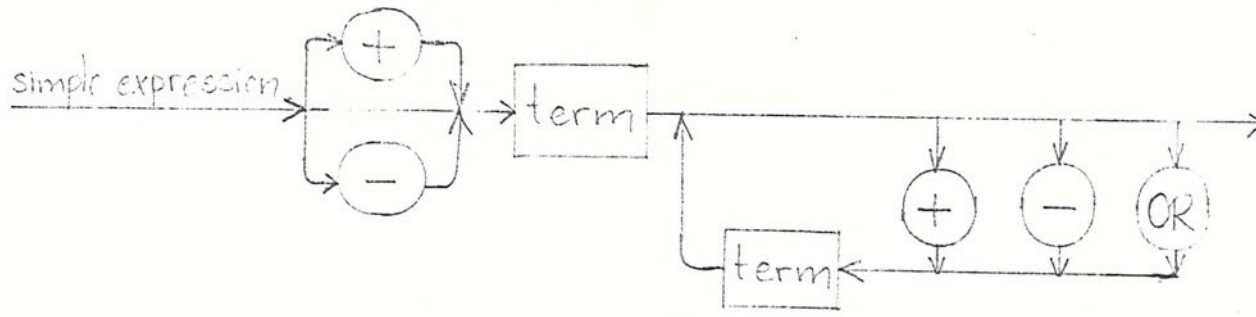


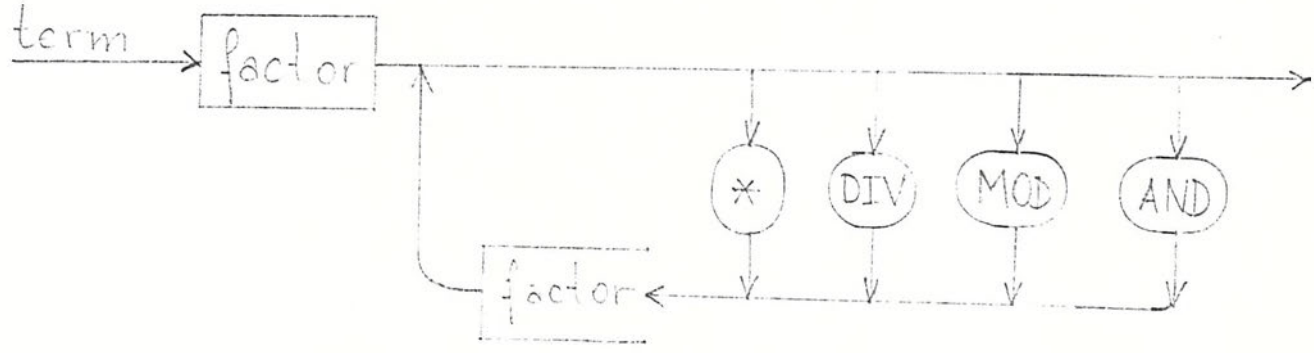


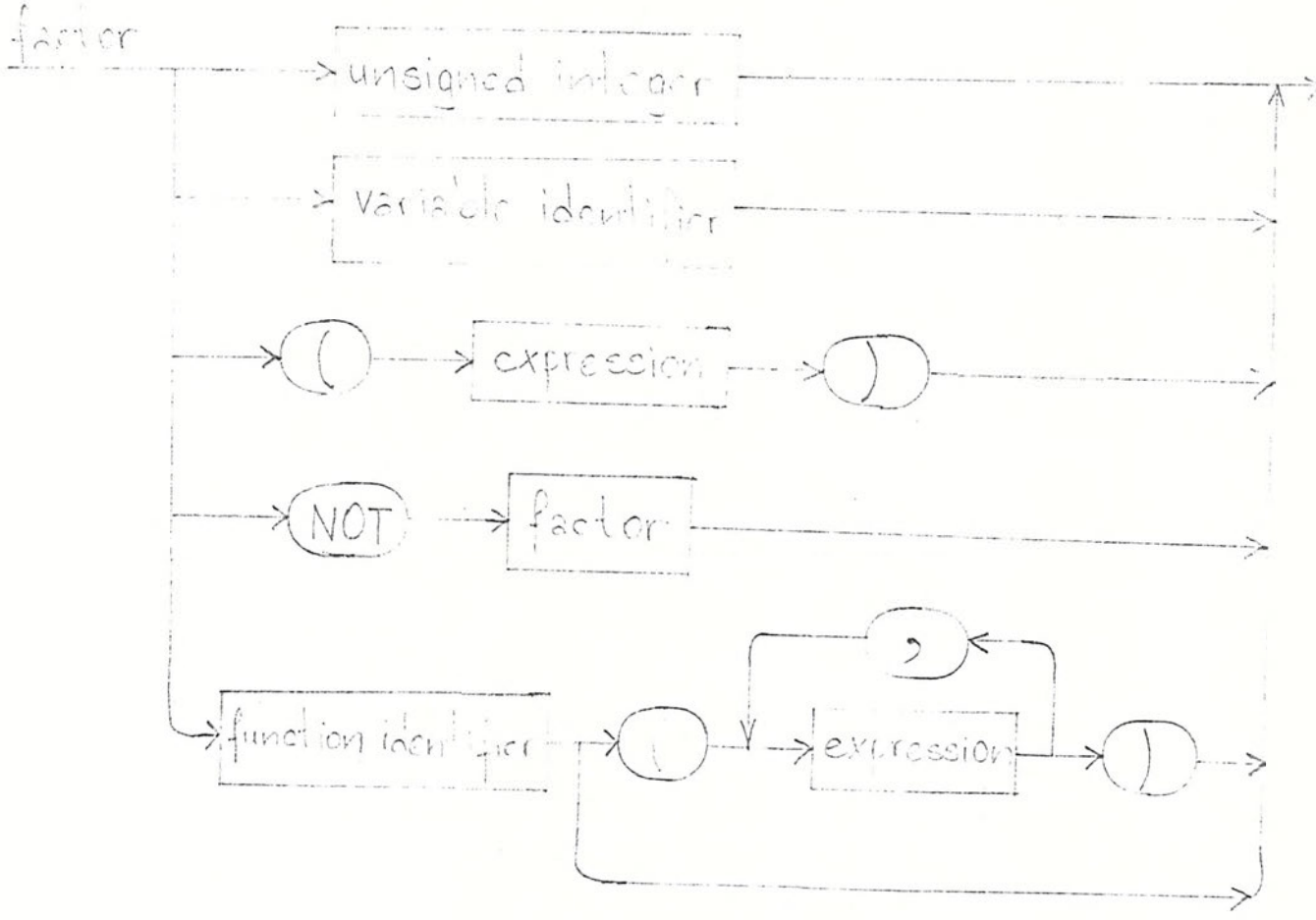
*not for functions.


statement
variable identifier
:=
expression
function identifier
procedure identifier
(
,
)
BEGIN
statement
END
;
IF
expression
THEN
statement
ELSE
statement
WHILE
expression
DO
statement


expression
simple expression
=
<>
<
<=
>=
>
simple expression


simple expression
+
−
term
+
−
OR
term


term
factor
*
DIV
MOD
AND
factor


factor
unsigned integer
variable identifier
(
expression
)
NOT
factor
function identifier
(
,
expression
)

APPENDIX II

BURNER-CONTROLLER IN PASCAL-MINUS

```
 1  PROGRAM BURNERCONTROLLER(INPUT,OUTPUT);
 2
 3    TYPE
 4      KEYBOARD=(D0,D1,D2,D3,D4,D5,D6,D7,D8,D9,RLOCK,EREAL,EON,EOFF,
 5                LOAD,MON,MOFF);
 6      DEVICE=(ACTIVE,PASSIVE);
 7
 8    VAR
 9      KEY,STATUS:KEYBOARD;
10      GAS,SPARK,MANUAL:DEVICE;
11      UP,PRESSED:BOOLEAN;
12      REAL,ON,OFF,DISPLAY,CHECKSUM,ATTEMPT:INTEGER;
13
14
15    FUNCTION NEWKEY:BOOLEAN;
16    BEGIN
17    END;
18
19    FUNCTION FLAME:BOOLEAN;
20    BEGIN
21    END;
22
23    FUNCTION PROGMEM:BOOLEAN;
24    BEGIN
25    END;
26
27
28    FUNCTION ISTIMED:BOOLEAN;
29    BEGIN
30      { ISTIMED:=((ON<=REAL) AND (REAL<=OFF)) OR }
31              { ((OFF<ON) AND ((ON<=REAL) OR (REAL<=OFF))) }
32    END;
33
34
35    FUNCTION ADD(A,B:INTEGER):INTEGER;
36      VAR SUM:INTEGER;
37      { A <= 2359 AND B <= 2359 }
38    BEGIN
39      SUM:= A MOD 100 + B MOD 100;
40      IF SUM > 59 THEN SUM:=SUM+40;
41      SUM:=SUM+(A DIV 100+B DIV 100)*100;
42      IF SUM > 2359 THEN SUM:=SUM-2400;
43      ADD:=SUM
44    END;   { ADD <= 2359 }
45
46    FUNCTION REALTIME:INTEGER;
47    BEGIN
48    END;
49
50    FUNCTION UPDATED:INTEGER;
51    BEGIN
52      { UPDATED:=ADD(REAL,ADD(ON,OFF)) }
53    END;
```

continued overleaf

```
55    FUNCTION REQUIRED:BOOLEAN;
56    BEGIN
57      { IF ISTIMED THEN }
58        { BEGIN REQUIRED:=TRUE; MANUAL:=PASSIVE END }
59      { ELSE }
60        { IF MANUAL=ACTIVE THEN REQUIRED:=TRUE }
61        { ELSE REQUIRED:=FALSE }
62    END;
63
64    PROCEDURE IGNITE;
65      { NOT ( PROGRAM-LED OR DATA-LED OR IGNITION-LED OR FLAME ) }
66    BEGIN
67      IF ATTEMPT < 5 THEN
68        BEGIN
69          GAS:=ACTIVE; SPARK:=ACTIVE;
70          IF NOT (FLAME) THEN BEGIN GAS:=PASSIVE; ATTEMPT:=ATTEMPT+1 END
71          ELSE ATTEMPT:=0;
72          { FLAME <=> (GAS=ACTIVE) AND (ATTEMPT=0) }
73          SPARK:=PASSIVE
74        END
75      ELSE DISPLAY:=2402   {IGNITION-LED AND (GAS=PASSIVE ) }
76    END:    { NOT ( PROGRAM-LED OR DATA-LED ) AND }
77            { IGNITION-LED AND ATTEMP=5 AND GAS=PASSIVE OR }
78       { NOT IGNITION-LED AND (FLAME AND ATTEMPT=0 OR GAS=PASSIVE AND ATTEMPT>0) }
79
80    PROCEDURE TURNOFF;
81    BEGIN
82      GAS:=PASSIVE; SPARK:=PASSIVE;
83      ATTEMPT:=0; MANUAL:=PASSIVE
84    END;
85
86    PROCEDURE INITIALIZE;
87      { NOT PROGRAM-LED }
88    BEGIN
89      REAL:=800; ON:=1000; OFF:=1400; CHECKSUM:=8080; DISPLAY:=800;
90      STATUS:=EREAL; PRESSED:=FALSE; UP:=TRUE; TURNOFF
91    END:    { NOT (PROGRAM-LED OR DATA-LED ) AND GAS=PASSIVE }
92
93
94    PROCEDURE RESETLOCKOUT;
95    { NOT(PROGRAM-LED OR DATA-LED) AND (GAS=PASSIVE) }
96    BEGIN
97      DISPLAY:=REAL; STATUS:=EREAL;
98      PRESSED:=FALSE; ATTEMPT:=0
99      { NOT ( PROGRAM-LED OR DATA-LED OR IGNITION-LED ) }
100   END;
101
```

continued overleaf

```
102   PROCEDURE TO(K:KEYBOARD);
103     { NOT ( PROGRAM-LED OR DATA-LED OR IGNITION-LED ) }
104
105     PROCEDURE SHIFTDISPLAY;
106     BEGIN
107       DISPLAY:=(DISPLAY-(DISPLAY DIV 1000)*1000)*10;
108       IF K=09 THEN DISPLAY:=DISPLAY+9
109       ELSE
110         IF K=08 THEN DISPLAY:=DISPLAY+8
111         ELSE
112          IF K=07 THEN DISPLAY:=DISPLAY+7
113          ELSE
114           IF K=06 THEN DISPLAY:=DISPLAY+6
115           ELSE
116            IF K=05 THEN DISPLAY:=DISPLAY+5
117            ELSE
118             IF K=04 THEN DISPLAY:=DISPLAY+4
119             ELSE
120              IF K=03 THEN DISPLAY:=DISPLAY+3
121              ELSE
122               IF K=02 THEN DISPLAY:=DISPLAY+2
123               ELSE
124                IF K=01 THEN DISPLAY:=DISPLAY+1
125     END;
126
127   BEGIN
128     IF (K=EREAL) OR (K=EON) OR (K=EOFF) THEN
129       BEGIN
130         STATUS:=K; PRESSED:=TRUE;
131         IF K=EREAL THEN DISPLAY:=REAL
132         ELSE
133           IF K=EON THEN DISPLAY:=ON
134           ELSE
135             IF K=EOFF THEN DISPLAY:=OFF
136       END
137     ELSE
138       IF (K=00) OR (K=01) OR (K=02) OR (K=03) OR (K=04) OR (K=05) OR
139          (K=06) OR (K=07) OR (K=08) OR (K=09) THEN
140         IF PRESSED THEN SHIFTDISPLAY
141                         {ENTER NEW DIGIT FROM THE RIGHT OF DISPLAY}
142         ELSE DISPLAY:=2403   {ERROR:DIGIT NOT PRECEDED WITH FUNCTION-KEY}
143       ELSE
144         IF K=LOAD THEN
145           BEGIN
146             IF PRESSED AND  ( DISPLAY < 2400 ) THEN
147               BEGIN
148                 IF STATUS=EREAL THEN REAL:=DISPLAY
149                 ELSE
150                   IF STATUS=EON THEN ON:=DISPLAY
151                   ELSE
152                     IF STATUS=EOFF THEN OFF:=DISPLAY;
153                 DISPLAY:=REAL; STATUS:=EREAL;
154                 CHECKSUM:=(REAL+ON+OFF) MOD 2400
155               END
156             ELSE DISPLAY:=2403;   {ERROR:ILLEGAL LOADING ATTEMPT}
157             PRESSED:=FALSE
158           END
159         ELSE
160           IF K=MON THEN MANUAL:=ACTIVE
161           ELSE
162             IF K=MOFF THEN MANUAL:=PASSIVE
163   END;   { NOT ( PROGRAM-LED OR DATA-LED OR IGNITION-LED ) }
```

continued overleaf

```
165     PROCEDURE IODUMMY(K:KEYBOARD);
166     BEGIN
167     END;
168
169
170
171   BEGIN{BURNERCONTROLLER}
172   { ALL VARIABLES UNDEFINED }
173     IF NOT (PROGMEM) THEN DISPLAY:=2400   { PROGRAM-LED }
174     ELSE
175       { NOT PROGRAM-LED }
176       BEGIN
177         INITIALIZE;   { ALL VARIABLES INITIALISED }
178         WHILE UP DO
179           { NOT ( PROGRAM-LED OR DATA-LED ) }
180           BEGIN
181             REAL:=REALTIME; CHECKSUM:=UPDATED;
182             IF DISPLAY <> 2402 THEN
183               { NOT ( PROGRAM-LED OR DATA-LED OR IGNITION-LED ) }
184               BEGIN
185                 DISPLAY:=REAL;
186                 IF REQUIRED THEN
187                   { ISTIMED OR ( MANUAL=ACTIVE ) }
188                   IF (GAS=ACTIVE) AND FLAME THEN ATTEMPT:=0
189                   ELSE IGNITE
190                     { FLAME (=) (GAS=ACTIVE) AND (ATTEMPT=0) }
191                 ELSE
192                   IF GAS=ACTIVE THEN
193                     TURNOFF   { (GAS=PASSIVE) AND (ATTEMPT=0) } ;
194                 IF NEWKEY THEN IODUMMY(KEY)
195               END
196               { NOT(PROGRAM-LED OR DATA-LED) }
197             ELSE
198               { NOT(PROGRAM-LED OR DATA-LED) AND IGNITION-LED }
199               IF NEWKEY AND (KEY=RLOCK) THEN
200                 RESETLOCKOUT {NOT(PROGRAM-LED OR DATA-LED OR IGNITION-LED)};
201               { NOT(PROGRAM-LED OR DATA-LED) }
202             IF CHECKSUM <> UPDATED THEN
203               { NOT(PROGRAM-LED) AND DATA-LED }
204               BEGIN
205                 DISPLAY:=2401; GAS:=PASSIVE; UP:=FALSE   { DATA-LED }
206               END
207           END
208           { NOT ( PROGRAM-LED ) AND DATA-LED }
209       END
210       { PROGRAM-LED OR DATA-LED }
211   END{BURNERCONTROLLER};
```

BURNER-

CONTROLLER IN FDL

```
                'TITLE' PROGRAM_BURNERCO;
                'TYPE' DEVICE;
                'CONSTANT' PASSIVE,ACTIVE:DEVICE;
                'TYPE' KEYBOARD;
                'CONSTANT' EREAL,RLOCK:KEYBOARD;
                'VARIABLE' ATTEMPT:INTEGER;
                'VARIABLE' CHECKSUM:INTEGER;
                'VARIABLE' DISPLAY:INTEGER;
                'VARIABLE' FLAME##0:BOOLEAN;
                'VARIABLE' GAS:DEVICE;
                'VARIABLE' KEY:KEYBOARD;
                'VARIABLE' MANUAL:DEVICE;
                'VARIABLE' NEWKEY##0:BOOLEAN;
                'VARIABLE' OFF:INTEGER;
                'VARIABLE' ON:INTEGER;
                'VARIABLE' PRESSED:BOOLEAN;
                'VARIABLE' PROGMEM##0:BOOLEAN;
                'VARIABLE' REAL:INTEGER;
                'VARIABLE' REALTIME##0:INTEGER;
                'VARIABLE' REQUIRED##0:BOOLEAN;
                'VARIABLE' SPARK:DEVICE;
                'VARIABLE' STATUS:KEYBOARD;
                'VARIABLE' UP:BOOLEAN;
                'VARIABLE' UPDATED##0:INTEGER;
                'VARIABLE' K#15:KEYBOARD;

{ 1}             'START'
{ 2}             'IF'PROGMEM##0
                 'THEN''GOTO' 2;
{ 3}             DISPLAY:= 2400;
{ 4}             'GOTO' 1;
{ 5}          2 :REAL:= 800;
{ 6}             ON:= 1000;
{ 7}             OFF:= 1400;
{ 8}             CHECKSUM:= 800;
{ 9}             DISPLAY:= 800;
{ 10}            STATUS:=EREAL;
{ 11}            PRESSED:=FALSE;
{ 12}            UP:=TRUE;
{ 13}            GAS:=PASSIVE;
{ 14}            SPARK:=PASSIVE;
{ 15}            ATTEMPT:= 0;
{ 16}            MANUAL:=PASSIVE;
{ 17}         3 :'IF' NOT (UP)
                 'THEN''GOTO' 1;
{ 18}            REAL:=REALTIME##0;
{ 19}            CHECKSUM:=UPDATED##0;
{ 20}            'IF'DISPLAY= 2402
                 'THEN''GOTO' 5;
{ 21}            DISPLAY:=REAL;
{ 22}            'IF' NOT (REQUIRED##0)
                 'THEN''GOTO' 7;
{ 23}            'IF'FLAME##0 AND
                     (GAS=ACTIVE)
                 'THEN''GOTO' 8;
{ 24}            'IF'ATTEMPT>= 5
                 'THEN''GOTO' 10;
{ 25}            GAS:=ACTIVE;
{ 26}            SPARK:=ACTIVE;
{ 27}            'IF'FLAME##0
                 'THEN''GOTO' 12;
{ 28}            GAS:=PASSIVE;
{ 29}            ATTEMPT:= 1+ATTEMPT;
{ 30}            'GOTO' 13;
{ 31}        12 :ATTEMPT:= 0;
{ 32}        13 :SPARK:=PASSIVE;
{ 33}            'GOTO' 8;
{ 34}        10 :DISPLAY:= 2402;
{ 35}         8 :'GOTO' 14;
{ 36}         7 :'IF'GAS<>ACTIVE
                 'THEN''GOTO' 14;
{ 37}            GAS:=PASSIVE;
{ 38}            SPARK:=PASSIVE;
{ 39}            ATTEMPT:= 0;
{ 40}            MANUAL:=PASSIVE;
{ 41}        14 :'IF' NOT (NEWKEY##0)
                 'THEN''GOTO' 6;
{ 42}            K#15:=KEY;
{ 43}         6 :'GOTO' 15;
{ 44}         5 :'IF' NOT (NEWKEY##0)
                  OR (KEY<>RLOCK)
                 'THEN''GOTO' 15;
{ 45}            DISPLAY:=REAL;
{ 46}            STATUS:=EREAL;
{ 47}            PRESSED:=FALSE;
{ 48}            ATTEMPT:= 0;
{ 49}        15 :'IF'CHECKSUM=UPDATED##0
                 'THEN''GOTO' 4;
{ 50}            DISPLAY:= 2401;
{ 51}            UP:=FALSE;
{ 52}            'GOTO' 1;
{ 53}         4 :'GOTO' 3;
{ 54}         1 :'FINISH'
```

```
'TITLE' PROGRAM_BURNERCO;
'TYPE' DEVICE;
'CONSTANT' PASSIVE,ACTIVE:DEVICE;
'TYPE' KEYBOARD;
'CONSTANT' EREAL,RLOCK:KEYBOARD;
'VARIABLE' ATTEMPT:INTEGER;
'VARIABLE' CHECKSUM:INTEGER;
'VARIABLE' DISPLAY:INTEGER;
'VARIABLE' FLAME##0:BOOLEAN;
'VARIABLE' GAS:DEVICE;
'VARIABLE' KEY:KEYBOARD;
'VARIABLE' MANUAL:DEVICE;
'VARIABLE' NEWKEY##0:BOOLEAN;
'VARIABLE' OFF:INTEGER;
'VARIABLE' ON:INTEGER;
'VARIABLE' PRESSED:BOOLEAN;
'VARIABLE' PROGMEM##0:BOOLEAN;
'VARIABLE' REAL:INTEGER;
'VARIABLE' REALTIME##0:INTEGER;
'VARIABLE' REQUIRED##0:BOOLEAN;
'VARIABLE' SPARK:DEVICE;
'VARIABLE' STATUS:KEYBOARD;
'VARIABLE' UP:BOOLEAN;
'VARIABLE' UPDATED##0:INTEGER;
'VARIABLE' K#15:KEYBOARD;

{ 1}      'START'
{ 2}      'IF' NOT (PROGMEM##0)
          'THEN'
             DISPLAY:= 2400 &
             'GOTO' 1
          'ELSE'
          'IF'PROGMEM##0
          'THEN'
             'MAP'
                ATTEMPT:= 0;
                CHECKSUM:= 800;
                DISPLAY:= 800;
                GAS:=PASSIVE;
                MANUAL:=PASSIVE;
                OFF:= 1400;
                ON:= 1000;
                PRESSED:=FALSE;
                REAL:= 800;
                SPARK:=PASSIVE;
                STATUS:=EREAL;
                UP:=TRUE;
             'END' &
             'GOTO' 3;
{ 17}     3 :'IF' NOT (UP)
          'THEN''GOTO' 1
          'ELSE'
          'IF'UP AND
              (DISPLAY= 2402)
          'THEN'
             'MAP'
                CHECKSUM:=UPDATED##0;
                REAL:=REALTIME##0;
             'END' &
             'GOTO' 5
          'ELSE'
          'IF'UP AND
              (DISPLAY<> 2402) AND
               NOT (REQUIRED##0)
          'THEN'
             'MAP'
                CHECKSUM:=UPDATED##0;
                DISPLAY:=REALTIME##0;
                REAL:=REALTIME##0;
             'END' &
             'GOTO' 7
          'ELSE'
          'IF'UP AND
              (DISPLAY<> 2402) AND
              REQUIRED##0 AND
               NOT (FLAME##0) AND
              (ATTEMPT<5)
           OR UP AND
              (DISPLAY<> 2402) AND
              REQUIRED##0 AND
              (GAS<>ACTIVE) AND
              (ATTEMPT<5)
          'THEN'
             'MAP'
                CHECKSUM:=UPDATED##0;
                DISPLAY:=REALTIME##0;
                GAS:=ACTIVE;
                REAL:=REALTIME##0;
                SPARK:=ACTIVE;
             'END'
```

BURNER-CONTROLLER IN FDL REDUCED TO 9 NODES

continued overleaf

```
                'ELSE'
                'IF'UP AND
                    (DISPLAY<> 2402) AND
                    REQUIRED##0 AND
                    FLAME##0 AND
                    (GAS=ACTIVE)
                'THEN'
                   'MAP'
                      CHECKSUM:=UPDATED##0;
                      DISPLAY:=REALTIME##0;
                      REAL:=REALTIME##0;
                   'END' &
                   'GOTO' 14
                'ELSE'
                'IF'UP AND
                    (DISPLAY<> 2402) AND
                    REQUIRED##0 AND
                     NOT (FLAME##0) AND
                    (ATTEMPT>= 5)
                 OR UP AND
                    (DISPLAY<> 2402) AND
                    REQUIRED##0 AND
                    (GAS<>ACTIVE) AND
                    (ATTEMPT>= 5)
                'THEN'
                   'MAP'
                      CHECKSUM:=UPDATED##0;
                      DISPLAY:= 2402;
                      REAL:=REALTIME##0;
                   'END' &
                   'GOTO' 14;
{ 27}           'IF' NOT (FLAME##0)
                'THEN'
                   'MAP'
                      ATTEMPT:= 1+ATTEMPT;
                      GAS:=PASSIVE;
                      SPARK:=PASSIVE;
                   'END' &
                   'GOTO' 14
                'ELSE'
                'IF'FLAME##0
                'THEN'
                   'MAP'
                      ATTEMPT:= 0;
                      SPARK:=PASSIVE;
                   'END' &
                   'GOTO' 14;
{ 36}        7 :'IF'GAS<>ACTIVE
                'THEN''GOTO' 14
                'ELSE'
                'IF'GAS=ACTIVE
                'THEN'
                   'MAP'
                      ATTEMPT:= 0;
                      GAS:=PASSIVE;
                      MANUAL:=PASSIVE;
                      SPARK:=PASSIVE;
                   'END' &
                   'GOTO' 14;
{ 41}       14 :'IF' NOT (NEWKEY##0)
                'THEN''GOTO' 15
                'ELSE'
                'IF'NEWKEY##0
                'THEN'
                   K#15:=KEY &
                   'GOTO' 15;
{ 44}        5 :'IF' NOT (NEWKEY##0)
                 OR (KEY<>RLOCK)
                'THEN''GOTO' 15
                'ELSE'
                'IF'NEWKEY##0 AND
                    (KEY=RLOCK)
                'THEN'
                   'MAP'
                      ATTEMPT:= 0;
                      DISPLAY:=REAL;
                      PRESSED:=FALSE;
                      STATUS:=EREAL;
                   'END' &
                   'GOTO' 15;
{ 49}       15 :'IF'CHECKSUM<>UPDATED##0
                'THEN'
                   'MAP'
                      DISPLAY:= 2401;
                      UP:=FALSE;
                   'END' &
                   'GOTO' 1
                'ELSE'
                'IF'CHECKSUM=UPDATED##0
                'THEN''GOTO' 3;
{ 54}        1 :'FINISH'
```

APPENDIX III

BURNER-CONTROLLER SYMBOLICLY EXECUTED, WITH CORRESPONDING LINE NUMBERS AND ASSERTIONS IN THE ORIGINAL PROGRAM

```
DEPARTMENT OF ELECTRONICS, UNIVERSITY OF SOUTHAMPTON.
SPADE SYMBOLIC EXECUTION.

------------------------  PATHLENGTH   1  ------------------------

NODE    2 PATH  1  (PREDECESSOR    1 PATH  1)

          TRAVERSAL CONDITION :
             TRUE
          PATH FUNCTION :
             UNIT FUNCTION

------------------------  PATHLENGTH   2  ------------------------

NODE    3 PATH  1  (PREDECESSOR    2 PATH  1)

          TRAVERSAL CONDITION :                    Line 177
              NOT ( NOT (PROGMEM##0))
          PATH FUNCTION :
             ATTEMPT:= 0
             CHECKSUM:= 800
             DISPLAY:= 800
             GAS:=PASSIVE
             MANUAL:=PASSIVE
             OFF:= 1400
             ON:= 1000
             PRESSED:=FALSE
             REAL:= 800
             SPARK:=PASSIVE
             STATUS:=EREAL
             UP:=TRUE

NODE    9 PATH  1  (PREDECESSOR    2 PATH  1)

          TRAVERSAL CONDITION :                    Line 173
              NOT (PROGMEM##0)
          PATH FUNCTION :
             DISPLAY:= 2400                        {PROGRAM-LED}
```

---------- PATHLENGTH 3 ----------

NODE 4 PATH 1 (PREDECESSOR 3 PATH 1)

Line 69

```
TRAVERSAL CONDITION :
    NOT ( NOT (PROGMEM##0)) AND
        (TRUE AND
        ( 800<> 2402) AND
        REQUIRED##0 AND
         NOT (FLAME##0) AND
        ( 0< 5)
    OR TRUE AND
        ( 800<> 2402) AND
        REQUIRED##0 AND
        (PASSIVE<>ACTIVE) AND
        ( 0< 5))
PATH FUNCTION :
    ATTEMPT:= 0
    CHECKSUM:=UPDATED##0
    DISPLAY:=REALTIME##0
    GAS:=ACTIVE
    MANUAL:=PASSIVE
    OFF:= 1400
    ON:= 1000
    PRESSED:=FALSE
    REAL:=REALTIME##0
    SPARK:=ACTIVE
    STATUS:=EREAL
    UP:=TRUE
```

{NOT (PROGRAM-LED OR DATA-LED OR IGNITION-LED)}

NODE 5 PATH 1 (PREDECESSOR 3 PATH 1)

Line 191

```
TRAVERSAL CONDITION :
    NOT ( NOT (PROGMEM##0)) AND
        TRUE AND
        ( 800<> 2402) AND
         NOT (REQUIRED##0)
PATH FUNCTION :
    ATTEMPT:= 0
    CHECKSUM:=UPDATED##0
    DISPLAY:=REALTIME##0
    GAS:=PASSIVE
    MANUAL:=PASSIVE
    OFF:= 1400
    ON:= 1000
    PRESSED:=FALSE
    REAL:=REALTIME##0
    SPARK:=PASSIVE
    STATUS:=EREAL
    UP:=TRUE
```

{NOT (PROGRAM-LED OR DATA-LED OR IGNITION-LED)}

```
---------------------- PATHLENGTH   4 ----------------------

NODE    6 PATH  1  (PREDECESSOR    4 PATH  1)

          TRAVERSAL CONDITION :
              NOT ( NOT (PROGMEM##0)) AND
                  (TRUE AND
                  ( 800<> 2402) AND
                  REQUIRED##0 AND
                   NOT (FLAME##0) AND
                  ( 0< 5)
              OR TRUE AND
                  ( 800<> 2402) AND
                  REQUIRED##0 AND
                  (PASSIVE<>ACTIVE) AND
                  ( 0< 5)) AND
                   NOT (FLAME##0)
          PATH FUNCTION :
             ATTEMPT:= 1+ 0
             CHECKSUM:=UPDATED##0
             DISPLAY:=REALTIME##0
             GAS:=PASSIVE
             MANUAL:=PASSIVE
             OFF:= 1400
             ON:= 1000
             PRESSED:=FALSE
             REAL:=REALTIME##0
             SPARK:=PASSIVE
             STATUS:=EREAL
             UP:=TRUE

          TRAVERSAL CONDITION :
              NOT ( NOT (PROGMEM##0)) AND
                  (TRUE AND
                  ( 800<> 2402) AND
                  REQUIRED##0 AND
                   NOT (FLAME##0) AND
                  ( 0< 5)
              OR TRUE AND
                  ( 800<> 2402) AND
                  REQUIRED##0 AND
                  (PASSIVE<>ACTIVE) AND
                  ( 0< 5)) AND
                   NOT ( NOT (FLAME##0))
          PATH FUNCTION :
             ATTEMPT:= 0
             CHECKSUM:=UPDATED##0
             DISPLAY:=REALTIME##0
             GAS:=ACTIVE
             MANUAL:=PASSIVE
             OFF:= 1400
             ON:= 1000
             PRESSED:=FALSE
             REAL:=REALTIME##0
             SPARK:=PASSIVE
             STATUS:=EREAL
             UP:=TRUE
```

Line 70

{GAS=PASSIVE AND ATTEMPT>0}

Line 71

{FLAME <=> (GAS=ACTIVE) AND (ATTEMPT=0)}

```
------------------------  PATHLENGTH   5  ------------------------
NODE   8 PATH  1  (PREDECESSOR   6 PATH  1)

        TRAVERSAL CONDITION :  Line 195 (through Line 70)
            NOT ( NOT (PROGMEM##0)) AND
                (TRUE AND
                ( 800<> 2402) AND
                REQUIRED##0 AND
                 NOT (FLAME##0) AND
                ( 0< 5)
            OR TRUE AND
                ( 800<> 2402) AND
                REQUIRED##0 AND
                (PASSIVE<>ACTIVE) AND
                ( 0< 5)) AND
                 NOT (FLAME##0) AND
                 NOT (NEWKEY##0)
        PATH FUNCTION :                  {NOT (PROGRAM-LED
           ATTEMPT:= 1+ 0                 OR DATA-LED)}
           CHECKSUM:=UPDATED##0
           DISPLAY:=REALTIME##0
           GAS:=PASSIVE
           MANUAL:=PASSIVE
           OFF:= 1400
           ON:= 1000
           PRESSED:=FALSE
           REAL:=REALTIME##0
           SPARK:=PASSIVE
           STATUS:=EREAL
           UP:=TRUE

        TRAVERSAL CONDITION :  Line 195 (through Line 71)
            NOT ( NOT (PROGMEM##0)) AND
                (TRUE AND
                ( 800<> 2402) AND
                REQUIRED##0 AND
                 NOT (FLAME##0) AND
                ( 0< 5)
            OR TRUE AND
                ( 800<> 2402) AND
                REQUIRED##0 AND
                (PASSIVE<>ACTIVE) AND
                ( 0< 5)) AND
                 NOT ( NOT (FLAME##0)) AND
                 NOT (NEWKEY##0)
        PATH FUNCTION :                  {NOT(PROGRAM-LED
           ATTEMPT:= 0                    OR DATA-LED)}
           CHECKSUM:=UPDATED##0
           DISPLAY:=REALTIME##0
           GAS:=ACTIVE
           MANUAL:=PASSIVE
           OFF:= 1400
           ON:= 1000
           PRESSED:=FALSE
           REAL:=REALTIME##0
           SPARK:=PASSIVE
           STATUS:=EREAL
           UP:=TRUE
```

---------- PATHLENGTH 6 ----------

```
NODE   3 PATH  1  (PREDECESSOR   8 PATH  1)

        TRAVERSAL CONDITION :
            NOT ( NOT (PROGMEM##0)) AND
               (TRUE AND
               ( 800<> 2402) AND
               REQUIRED##0 AND
                NOT (FLAME##0) AND
               ( 0< 5)
            OR TRUE AND
               ( 800<> 2402) AND
               REQUIRED##0 AND
               (PASSIVE<>ACTIVE) AND
               ( 0< 5)) AND
                NOT (FLAME##0) AND
                NOT (NEWKEY##0) AND
               (UPDATED##0=UPDATED##0)
        PATH FUNCTION :
           ATTEMPT:= 1+ 0
           CHECKSUM:=UPDATED##0
           DISPLAY:=REALTIME##0
           GAS:=PASSIVE
           MANUAL:=PASSIVE
           OFF:= 1400
           ON:= 1000
           PRESSED:=FALSE
           REAL:=REALTIME##0
           SPARK:=PASSIVE
           STATUS:=EREAL
           UP:=TRUE

        TRAVERSAL CONDITION :
            NOT ( NOT (PROGMEM##0)) AND
               (TRUE AND
               ( 800<> 2402) AND
               REQUIRED##0 AND
                NOT (FLAME##0) AND
               ( 0< 5)
            OR TRUE AND
               ( 800<> 2402) AND
               REQUIRED##0 AND
               (PASSIVE<>ACTIVE) AND
               ( 0< 5)) AND
                NOT ( NOT (FLAME##0)) AND
                NOT (NEWKEY##0) AND
               (UPDATED##0=UPDATED##0)
        PATH FUNCTION :
           ATTEMPT:= 0
           CHECKSUM:=UPDATED##0
           DISPLAY:=REALTIME##0
           GAS:=ACTIVE
           MANUAL:=PASSIVE
           OFF:= 1400
           ON:= 1000
           PRESSED:=FALSE
           REAL:=REALTIME##0
           SPARK:=PASSIVE
           STATUS:=EREAL
           UP:=TRUE
```

Line 207 (through Line 70)

{NOT (PROGRAM-LED OR DATA-LED)}

Line 207 (through Line 71)

{NOT (PROGRAM-LED OR DATA-LED)}

```
NODE    9 PATH  3  (PREDECESSOR    8 PATH  1)

    TRAVERSAL CONDITION :
        NOT ( NOT (PROGMEM##0)) AND
            (TRUE AND
            ( 800<> 2402) AND
            REQUIRED##0 AND
             NOT (FLAME##0) AND
            ( 0< 5)
        OR TRUE AND
            ( 800<> 2402) AND
            REQUIRED##0 AND
            (PASSIVE<>ACTIVE) AND
            ( 0< 5)) AND
             NOT (FLAME##0) AND
             NOT (NEWKEY##0) AND
            (UPDATED##0<>UPDATED##0)
    PATH FUNCTION :
       ATTEMPT:= 1+ 0
       CHECKSUM:=UPDATED##0
       DISPLAY:= 2401
       GAS:=PASSIVE
       MANUAL:=PASSIVE
       OFF:= 1400
       ON:= 1000
       PRESSED:=FALSE
       REAL:=REALTIME##0
       SPARK:=PASSIVE
       STATUS:=EREAL
       UP:=FALSE
```

Line 205 (through Lines 189, 70, 194)

{DATA-LED}

```
NODE    3 PATH  2  (PREDECESSOR    8 PATH  2)

    TRAVERSAL CONDITION :
        NOT ( NOT (PROGMEM##0)) AND
            TRUE AND
            ( 800<> 2402) AND
             NOT (REQUIRED##0) AND
            (PASSIVE<>ACTIVE) AND
             NOT (NEWKEY##0) AND
            (UPDATED##0=UPDATED##0)
    PATH FUNCTION :
       ATTEMPT:= 0
       CHECKSUM:=UPDATED##0
       DISPLAY:=REALTIME##0
       GAS:=PASSIVE
       MANUAL:=PASSIVE
       OFF:= 1400
       ON:= 1000
       PRESSED:=FALSE
       REAL:=REALTIME##0
       SPARK:=PASSIVE
       STATUS:=EREAL
       UP:=TRUE
```

Line 207 (through Line 191)

{NOT (PROGRAM-LED OR DATA-LED)}