

THE STATIC DETECTION OF
SYNCHRONIZATION ANOMALIES
IN HAL/S PROGRAMS[†]

by

Guy Bristow
Department of Computer Science
University of Colorado at Boulder
Boulder, Colorado 80309

CU-CS-165-79

August, 1979

[†]This work was supported by
grant NSG 1476 from NASA
Langley Research Center.

THE STATIC DETECTION OF
SYNCHRONIZATION ANOMALIES
IN HAL/S PROGRAMS[†]

by

Guy Bristow
Department of Computer Science
University of Colorado at Boulder
Boulder, Colorado 80309

CU-CS-165-79

August, 1979

[†]This work was supported by
grant NSG 1476 from NASA
Langley Research Center.

Bristow, Guy Neil Rowland (M.S., Computer Science)

The Static Detection of Synchronisation Anomalies in HAL/S Programs

Thesis directed by Associate Professor William E. Riddle.

The introduction of concurrent languages has produced new classes of programming errors not found in purely sequential languages. One concurrent language, HAL/S, contains a wide variety of inter-process synchronisation features, with few restrictions on their use. Errors and potential errors can occur in particularly subtle and unusual ways, making them hard to avoid and hard to detect.

One approach to error detection is called anomaly detection. An anomaly is any situation in a program which, it is felt, represents a deviation from the developer's intent. In the anomaly detection system, therefore, the developer is presented with a list of anomalies which must be examined to determine which ones represent genuine errors.

This thesis defines a number of anomalies in HAL/S that arise directly from the synchronisation features in the language. Subsequently, the design of a system is presented that statically detects all such anomalies that are present in an arbitrary HAL/S program.

This abstract is approved as to form and content.

Signed

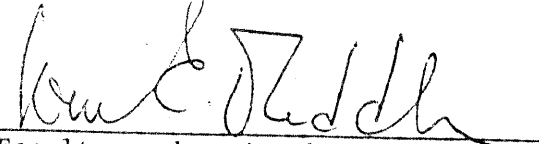

Faculty member in charge of thesis

TABLE OF CONTENTS

| CHAPTER | PAGE |
|--|------|
| I. INTRODUCTION | 1 |
| What is an anomaly? | 2 |
| Requirements of our system | 3 |
| Thesis layout | 3 |
| II. THE HAL/S LANGUAGE | 5 |
| General observations | 6 |
| Synchronisation in HAL/S | 6 |
| Synchronisation anomalies | 12 |
| III. GENERAL RESTRICTIONS | 15 |
| RTE-clock time and process priorities | 16 |
| Branches, loops, and cycles | 18 |
| Non synchronisation statements and variables | 19 |
| Restrictions | 21 |
| Complexity of the remaining problem | 22 |
| IV. EARLIER METHODS | 23 |
| Methods involving resource dependency cycles | 23 |
| Petri nets | 26 |
| V. THE INTER PROCESS PRECEDENCE GRAPH | 40 |
| Further simplification of the problem | 40 |
| Process flow graphs | 45 |
| Loops | 46 |
| Inter process precedence edges | 48 |

| CHAPTER | PAGE |
|--|------|
| VI. EXECUTION SEQUENCE SETS | 52 |
| Execution sequence sets | 52 |
| Generation of the execution sequence sets | 54 |
| Spurious IPPE elimination | 57 |
| Taking terminations into account | 60 |
| Properties of the execution sequence sets | 61 |
| VII. SYNCHRONISATION ANOMALY DETECTION | 66 |
| Infinite waits | 66 |
| Rescheduling a live process | 69 |
| Terminating, cancelling, or updating the priority of a dead process | 70 |
| Terminating an independent process | 71 |
| Premature termination | 71 |
| VIII. HOW GOOD IS THE SYSTEM? | 72 |
| Safe situations falsely identified as anomalous | 72 |
| IX. FUTURE WORK | 78 |
| Improvements to the system | 78 |
| Extensions to the system | 82 |
| Similar systems for other concurrent languages | 83 |
| X. CONCLUSIONS | 84 |
| APPENDICES | 87 |
| A. PROOF OF NP-COMPLETENESS | 87 |
| B. PSEUDO-CODE FOR THE SYSTEM | 89 |

ACKNOWLEDGEMENTS

I would like to thank the other members of the HAL/S team - William Riddle, Carol Drey, Brian Edwards, Lloyd Fosdick and John Humbrecht - for their good ideas and suggestions. In particular, some of section 6.3 and appendix C.2 was written by Carol Drey, and many of the other ideas presented here are due to her. I would also like to thank Lee Osterweil and Paul Zeiger for their help, and any other members of the Computer Science Dept. who deserve thanks, or feel they deserve thanks. Finally, I would like to thank SPL International, London, England, for their financial help.

The work presented in this thesis was supported by NASA Langley Research Centre, under grant NSG 1476.

CHAPTER I

INTRODUCTION

Much of the cost of producing software goes into debugging, testing, and verifying the correctness of that software. To reduce this cost, and to aid in the production of more reliable software, several systems have been developed to help detect errors and potential errors not normally checked for by compilers, and to (subsequently) verify the correctness of programs.

Our background and experience with the DAVE system [1] has shown that static data-flow analysis can be a powerful tool in the detection of certain classes of errors in FORTRAN programs. It was therefore logical for us to try and use similar techniques when we attempted to develop a system for error detection in HAL/S programs.

The HAL/S programming language [2], however, contains many powerful multi-processing features which encourage the use of concurrency in almost every non trivial program. This introduces a whole new class of programming errors not found in languages without such features, and which can occur anywhere that the concurrency features of HAL/S are used.

This thesis presents the work we have performed to develop a system for the static detection of anomalies arising directly from concurrency and the synchronisation constructs in HAL/S programs. It is intended to be used in conjunction with data-flow analysis techniques of the types found in DAVE.

1.1

WHAT IS AN ANOMALY?

Clearly the ideal tool for use in debugging and testing a program would uncover all of the errors in that program. Errors can be caused by the programmer having misunderstood what the program should achieve. To uncover these errors requires a total knowledge of how the program is intended to perform, and is not attempted by our system. Alternatively, errors can be caused by programmer 'mistakes'. Such errors represent genuine deviations from the programmer's intent, and it is in the detection of these errors that we aim our system.

Deciding what represents an anomaly is not as easy as defining what is an error. The errors we would like to detect are deviations from the programmer's intent, but without a complete knowledge of that intent we cannot guarantee to detect all such errors. What we can do, however, is to make various assumptions about the programmer's intent and to define any situations which violate these assumptions as anomalous.

The first assumption is that the programmer intends the program to be safe and reliable, and hence to contain no situations having the potential to cause run-time errors. We therefore define any situation having the potential to cause a run-time error as anomalous. The second assumption is that the programmer intended to write a 'normal' program that obeys 'good' programming principles. Accordingly we further define any situation representing a deviation from normal or good programming practices as anomalous. This type of anomaly is somewhat vague, and hence there is a very broad spectrum

of situations which are potentially anomalous to a greater or lesser extent.

1.2

REQUIREMENTS OF OUR SYSTEM

Before we started our work on the design of an anomaly detection system we initially established a number of properties we felt our system should possess. Firstly, it should be immediately applicable to the program under analysis, and not require the preliminary production of some parallel representation of that program. Secondly, since our system is intended to save time for the program developers, it should be simple and fast to use. It should not require the user to insert a large amount of additional information along with the program text. Thirdly, to be useful as a program verification aid, it should reliably detect all of the anomalies of the specified types in the program being analysed. Fourthly, it should have reasonable computational properties, which we have initially taken to mean polynomial time and space bounds. Finally, for it to be a useful time saving tool, we would like to maximise the percentage of reported anomalies that represent actual error situations in the program.

1.3

THESIS LAYOUT

The next chapter gives an introduction to the HAL/S language and the synchronisation facilities available in it. Chapter three contains restrictions and simplifications we have imposed on the analysis we can perform. Chapter four contains a brief look at our earlier approaches to the system. Chapters five, six, and seven

present the final system we have designed. Chapter eight discusses the quality of the results that our system produces. Chapter nine outlines some future improvements that could be made to the system, and also other areas in which similar systems could be used.

Our work has been closely related to that of Taylor and Osterweil [3], who have designed a system that is very similar in capabilities and approach to our own. In addition, work is being done by Reif [4] on the analysis of concurrent processes that utilise different synchronisation constructs to those available in HAL/S. Many references to background work appear in our earlier report [5].

CHAPTER II

THE HAL/S LANGUAGE

The HAL/S programming language was developed by Intermetrics Inc. specifically for the production of flight software on the NASA space shuttle program. It is a real-time control language, having blocks of code (called programs and tasks) which can be scheduled for execution in a variety of different ways. These programs and tasks, collectively referred to as processes, can run logically or actually in parallel, and can communicate using several different methods.

The language contains a large number of different data types and provides many operations for them. In addition, it allows for multi-dimensional arrays and tree-like structures. For a complete definition of the language see the HAL/S language definition [2].

Our interest in HAL/S lies in the static detection of errors and potential errors that are not checked for by the HAL/S compiler. In particular, this thesis focuses on those anomalies that arise specifically from concurrency and the synchronisation operations in programs. For a description of some other anomalies and static methods for their detection see e.g. [3] and [5].

Before entering a discussion of these anomalies we first give an introduction to the program structures and synchronisation constructs available in HAL/S. The other facilities and operations available in HAL/S are not strictly relevant to this report and are not discussed here.

2.1

GENERAL OBSERVATIONS

As mentioned, HAL/S contains facilities for running processes in parallel. The run-time counterpart of a program in a language without concurrency is called a program complex in HAL/S. A program complex consists of an arbitrary number of object modules combined together, each object module being the result of running the compiler with a single compilation unit.

There are four types of object modules, namely program, procedure, function, and compool modules. A program module is independently executable and comes from compiling a single program block. A program block in turn consists of a program, its associated tasks, and the data, procedures, and functions shared by the program and its tasks. Procedure and function modules come from compiling independent procedures and functions and can be called from any of the program modules. Compool modules contain shared data that is common to all the program, procedure, and function modules in the program complex.

One important point to note is that the scope rules in HAL/S are such as to prevent any recursion.

2.2

SYNCHRONISATION IN HAL/S

2.2.1 THE REAL-TIME EXECUTIVE

At run-time the job of controlling the execution of all the processes is handled by the Real-Time Executive, or RTE. The actual details as to how the RTE operates vary from implementation to implementation, but conceptually all RTEs behave alike.

The RTE maintains a process queue containing the current state of all processes that have been scheduled (see 2.2.3), but have not yet completed execution. For our purposes we consider that such processes are in one of three possible states:

i) A process is 'active' if it is currently being executed. In a multi-processor environment it may be possible for processes to run actually in parallel, in which case more than one process may be active at a given time.

ii) A process is 'suspended' if it is ready to execute in every respect, but is waiting for the availability of a processor. A process which is either active or suspended is said to be 'ready'.

iii) A process is 'stalled' if it is waiting for some (as yet) unsatisfied condition (see 2.2.3 and 2.2.4).

Each process has a priority associated with it. These priorities operate in the normal way in that if two or more processes are ready, the one with the highest priority will become active.

In addition, the RTE maintains a clock recording elapsed time (RTE-clock time) measured in units bearing an implementation dependent relationship to actual time.

2.2.2 EVENT EXPRESSIONS

Associated with each process is a boolean variable, called a process event, which has the same name as the process. This process event is maintained by the RTE, having the value true if the process is currently on the process queue (i.e. is active, suspended, or stalled) and false otherwise.

In addition, there is a type of boolean variable called an event variable, along with three operations (SET, RESET, and SIGNAL) that can be performed on it. Event variables come in two different forms - latched and unlatched.

A latched event variable is initialised to true or false by its declaration. The execution of a set statement on a latched event variable results in it being assigned the value true, regardless of its previous value. Similarly, the execution of a reset statement on a latched event variable results in it being assigned the value false. The execution of a signal statement on a latched event variable can be regarded as momentarily, but not permanently, reversing the value of the variable.

An unlatched event variable is always initialised to the value false. The operations set and reset are illegal on an unlatched event variable. The execution of a signal statement on an unlatched event variable can be regarded as momentarily setting its value to true, and then back to false.

An event expression is any logical expression containing process events, event variables, and the operators AND, OR, and NOT in the normal way.

2.2.3 THE SCHEDULE, CLOSE, AND RETURN STATEMENTS

Processes are placed on the process queue by means of the SCHEDULE statement. The statement allows for five optional ways to modify the conditions of execution of the scheduled process:

- i) The process may be scheduled such that it is initially stalled either for a specified RTE-clock time interval, or until a specified

RTE-clock time is reached, or until a specified event expression is true. If no such condition is specified, or the condition is true at the time of scheduling, the process is immediately placed in the ready state.

ii) The process can be given any specified priority. If no priority is specified, the priority of the scheduling process is assumed.

iii) The process can be specified to be dependent on the scheduling process (i.e. the process executing the schedule statement). In this case the scheduled process cannot remain on the process queue after the scheduling process has been removed. Note that all processes are ultimately dependent on their enclosing program.

iv) The process can be scheduled such that it will execute cyclically (i.e. it will repeat its execution) until some halting criterion is reached. It can be specified that the process will start each cycle as soon as the previous cycle has finished; or it can be specified that the process will enter the stalled state for a given RTE-clock time between each cycle; or it can be specified that the start of each cycle will occur at a given RTE-clock time interval. If there is no repetition specified the process will be executed at most once.

v) The process can be given a terminating criterion consisting of the satisfying of a specified event expression or the reaching of a specified RTE-clock time. If this criterion is satisfied at schedule time the process is not executed at all. Otherwise, for a process scheduled cyclically, the criterion is checked at the end of each cycle, and the process is removed from the process queue if the

criterion is satisfied.

The main program is not initiated by a schedule statement and must be added to the process queue in some other, implementation dependent, way.

A process normally completes a cycle by the execution of a CLOSE statement. In addition, RETURN statements can appear anywhere in the process, and are treated as branches to the close statement. If the process was not scheduled cyclically, or if its completion criterion is satisfied, the process will complete its execution when the close statement is executed. If the process has any dependent sons on the process queue at this point the process is stalled until all dependent sons have completed execution and been removed from the process queue before it too is removed. Otherwise, the process is immediately removed from the process queue.

2.2.4 OTHER SYNCHRONISATION STATEMENTS

A process can cancel either itself, or a specified list of dependent processes, by executing a CANCEL statement. If a process is cancelled before it has become active it is removed from the process queue immediately (i.e. it does not execute at all). Otherwise, a cancelled process that was scheduled cyclically will complete execution at the end of its current cycle.

The TERMINATE statement is used to immediately remove from the process queue either the process executing the statement or a specified list of dependent processes. When a process is terminated in this way, all of its dependent sons currently on the process queue are also immediately terminated, and removed from the process queue

in the same way.

A process can, at any time, update its own or any other process's priority by executing an UPDATE PRIORITY statement.

A process may place itself in the stalled state by executing a WAIT statement with a specified wait completion criterion. This criterion can be either the satisfying of a specified event expression or the reaching of a specified RTE-clock time. As soon as this criterion is satisfied the process is placed back in the ready state.

One other synchronisation construct is available in HAL/S, namely the UPDATE BLOCK. It is used to provide a controlled environment for accessing data shared by two or more processes. A variable can be declared locked, in which case it can only be accessed from within an update block. Should two or more processes attempt to enter update blocks containing assignment statements on the same locked variable, the first process to enter its update block completes execution of the block before the other process can enter its own block. Update blocks containing only references to the same shared locked variables can be executed in parallel. If one process only requires to reference a locked variable while another may attempt to assign to that variable, some safe overlapping may be allowed depending on the particular implementation. The only synchronisation statements allowed inside update blocks are signal, set, and reset statements.

2.3

SYNCHRONISATION ANOMALIES

Deciding just which situations are anomalous is an open question. We have possibly been somewhat cautious in our decisions and have identified a list of nine situations, arising directly from the synchronisation constructs in HAL/S, which we regard as anomalous. We do not claim that this list is exhaustive.

Of these nine anomalous situations, eight would potentially cause run-time errors. The ninth - premature termination of a process - does not itself cause a run-time error, but it may be indicative of a programming error.

i) Potentially infinite wait.

A run-time error results if, at any time, every process on the (non empty) process queue is stalled while waiting for conditions other than RTE-clock time conditions. Clearly in this situation, none of the processes can become active unless an event change occurs (i.e. the value of a process event or event variable changes), and no event changes can occur because no processes can execute. We extend this anomaly category further to include any situations in which a process or processes can be stalled for a potentially infinite amount of time. This anomaly category includes all potential deadlocks, where two or more processes are cyclically waiting for each other.

ii) Rescheduling a 'live' process.

It is illegal in HAL/S for a schedule statement to be executed on a process that is still on the process queue from a previous schedule.

iii) A process cycle time is too short.

A run-time error results if a process that is scheduled cyclically is due to start a new cycle before the previous cycle has completed. This can only occur if the process was scheduled such that the starts of consecutive cycles are at specified RTE-clock time intervals.

iv) Cancelling a 'dead' process.

A run-time error results if a cancel statement is executed on a process that is not on the process queue. This, and the next two anomalies, are not serious, and the action of the RTE may be to ignore errors of this type.

v) Updating the priority of a 'dead' process.

A run-time error results if an update priority statement is executed on a process not on the process queue.

vi) Terminating a 'dead' process.

A run-time error results if a terminate statement is executed on a process not on the process queue.

vii) Illegal priority.

A run-time error results if the priority of a process is given as, or updated to, a value that is inconsistent with the priority numbering scheme established for the particular implementation.

viii) Terminating an independent process.

It is illegal in HAL/S to terminate a process unless that process is a dependent son of the terminating process (the process executing the terminate statement).

ix) Premature termination.

Although it does not violate the rules of HAL/S, we regard the termination of a process prior to its normal completion as an anomaly and a potential programming error. For instance, if a process that updates a database is terminated prematurely, the database may be left in an inconsistent state.

CHAPTER III

GENERAL RESTRICTIONS

We believe that in the general case the problem of determining whether a given HAL/S program contains errors of the types specified in 2.3 is undecidable, although we have not proved this. Given a knowledge of the particular system on which the program will run the problem becomes decidable, because of the finite (but large) number of states in the system, but it is for all practical purposes computationally infeasible. Furthermore, to perform a strict analysis of a particular program requires a complete knowledge of the implementation and environment in which that program will run. Reasons for this will become apparent in the next section.

As stated, our aim has been to produce a system which is simple to use, has 'reasonable' computational time and space bounds, and will reliably detect the anomalies in a program. The first two of these requirements have forced us to accept, in the light of the complexity of the problem, that our system can only find an approximation to the actual set of the specified anomalies in a particular program. The final requirement implies that the anomalies we identify will include all of the actual anomalies, but must also inevitably include some situations we identify as anomalous which are in fact perfectly safe.

Accordingly, we decided to apply our algorithms to a simplification of the HAL/S program under analysis. The simplifications are as follows.

3.1 RTE-CLOCK TIME AND PROCESS PRIORITIES

To accurately take account of the relative order of execution of the statements in different processes requires a total knowledge of all factors that can affect that order. These factors would include:

The system configuration.

One would need to know how many processors are available to the program, the relative execution speeds and system overheads of these processors, other independent processes which may be sharing the processors, relative speed of data transfer between and among processors and devices etc.

Implementation details.

e.g., which processes will run on which processors, the range of legal priorities, methods of storing and accessing shared data, relative speed of each RTE operation, correspondence of RTE-clock time to relative execution speeds etc.

Outside effects.

e.g., expected time that inputs will arrive from independently running devices, anticipated value ranges for key input variables (where, for instance, an assigned priority or the number of cycles around a loop depend on an input value) etc.

One of the consequences of our decision to make the system simple to use is that we cannot ask the user to provide such a large amount of information. Therefore, we have decided to ignore process priorities and RTE-clock times for the purposes of the analysis.

Since we ignore process priorities, we assume that, at any given time, any of the ready processes could be active, and we check for anomalies in all the resulting execution sequences. We define an execution sequence as the ordered set of statements that would be executed during a run of the program. It is assumed that where sections of two or more processes execute concurrently, there will still be an order in time that can be applied to the execution of the individual statements; i.e., at some (possibly atomic) level no two actions can occur simultaneously. Clearly, among the execution sequences that we check for anomalies are all those that could actually occur when the program is run, so we detect, among others, the anomalies present in those sequences.

As we ignore RTE-clock time, we must ignore all situations in which a process is stalled while waiting for an RTE-clock time condition. In such situations we assume the process never leaves the ready state. The effect of such a wait is to force the continuing execution of other concurrent processes to occur between the statement preceding the wait and the statement following the wait. Since we check for anomalies in all execution sequences, clearly we check all sequences in which other concurrent processes would execute for arbitrary amounts of time between those two statements anyway. Therefore we find all anomalies that would be caused by the wait.

One effect of ignoring RTE-clock time is that we cannot detect anomalies of the type where a process is due to start a new cycle before the previous cycle has been completed. However, we feel that all such situations have the potential for error anyway (see 8.1), and we therefore give a warning message wherever such a situation appears.

In summary, the only forced execution ordering that we take account of in our analysis is where such an ordering is forced by the synchronisation statements, other than those involving RTE-clock time and process priorities.

3.2

BRANCHES, LOOPS, AND CYCLES

In many cases the choice of which path is taken at a conditional branch can depend upon input values, RTE-clock times, the execution sequence to date, etc. We have already accepted that this information will not in general be available, and so there will be situations where it is not possible to tell which branches can be taken.

Accordingly, we have decided to assume that, where a branch occurs in a process, any of the paths can be taken at that branch. We check for anomalies in all combinations of paths through all the processes in the program under analysis.

For the same reasons it is not possible, in general, to calculate how many times a particular loop in a process will be executed. We therefore assume that each loop can be executed any arbitrary number of times (greater than or equal to one for a loop that must be executed at least once, and greater than or equal to

zero otherwise).

Similarly, for a process that is scheduled with a repeat clause it is not possible, in general, to determine how many times that process will be executed. Again, we assume that any such process scheduled cyclically can be executed any arbitrary number of times greater than or equal to zero.

In summary, we assume that any path through each process is possible, where a path consists of making an arbitrary choice at each conditional branch, combined with an arbitrary finite (but unbounded) choice, at each loop encountered in that path, for the number of cycles around that loop. We must check for anomalies in all possible combinations of paths through the processes in the program under analysis. Clearly, some of these combinations are those that could occur when the program runs, so we detect anomalies in those combinations.

3.3 NON SYNCHRONISATION STATEMENTS AND VARIABLES

In general, it is not possible to determine the value of an arbitrary expression, since this may depend on input values etc. We therefore have to accept that we cannot always determine whether the priority assigned to a process is valid in a particular HAL/S implementation. We cannot therefore detect anomalies of the type where a priority is given as, or updated to, an illegal value.

The remaining anomalies that we are interested in arise directly from the synchronisation statements in the program being considered. The only effect that the other statements in the program can have on our analysis is in determining which paths can be taken

through each process (and hence which synchronisation statements can be executed) and when each point in the paths can be reached (and hence the possible orders that those statements can be executed).

However, we have already assumed that all paths through a process, and all execution sequences of the statements in concurrent processes, are possible. The non synchronisation statements do not therefore affect the analysis except in determining where branches and loops occur, and where paths can join, within processes. We therefore ignore all non synchronisation statements and variables and only retain this necessary information.

Extending this still further, we can ignore much of the information in the synchronisation statements themselves. For the schedule statement, we ignore any priority clause, any RTE-clock time conditions, and any terminating condition for a process scheduled cyclically (since we assume the process can execute any arbitrary number of times). The only information that needs to be retained is:

- the satisfying condition for a process scheduled such that it immediately enters the stalled state waiting for a specified event expression to become true.

- whether or not the scheduled process is dependent on the scheduling process.

- whether the process will cycle or not.

- whether the process has a terminating criterion specified. If so, it is assumed that the criterion can be true at the time the schedule statement is executed, and hence that the process may not execute even once.

Similarly, we can ignore any waits for RTE-clock time conditions, any update blocks, any cancel statements, and any update priority statements. In the latter two cases, it is necessary to remember where the statements occur, to check for the possibility of cancelling or updating the priority of a dead process.

3.4

RESTRICTIONS

There are two language features present in HAL/S which our analysis system cannot satisfactorily deal with. Such features cause similar problems in all static data-flow analysis systems for languages containing them. These features are arrays (and for our analysis, this implies arrays of event variables) and name variables.

The problem with arrays stems from the fact it is not possible, in general, to determine the value of an arbitrary expression. An array subscript in HAL/S can be any expression evaluating to an integer, and hence it is not always possible to determine, at an array reference, which element of the array is being referenced.

The problem is very similar with name variables. A name variable has a particular type associated with it and can be made equivalent to any variable of that particular type. A reference to a name variable implies a reference to the variable it is currently equivalent to. In addition to the normal data types in HAL/S, name variables can also be of types PROGRAM, TASK, PROCEDURE or FUNCTION. It is not always possible to determine, at a particular statement containing a reference to a name variable, which variable, (or program, task, procedure or function) that name variable is

equivalent to at that point.

This thesis does not address the issue of name variables and event variable arrays. The current analysis ignores all occurrences of such features in the program under analysis. The system therefore does not guarantee to find all anomalies of the specified types in programs containing arrays of event variables or name variables of types event variable, program, task, procedure, and function.

3.5

COMPLEXITY OF THE REMAINING PROBLEM

Given the restrictions and simplifications presented earlier in this chapter, the remaining problem we are tackling is at least well defined.

However, the problem is still far from simple. Appendix A contains a proof that the decidability of whether a particular program (containing only a small subset of the language features possible in the general remaining problem) contains an anomaly of the type where a process can be rescheduled while still on the process queue, is at least as hard as an NP-complete problem. The general remaining problem, containing as it does the potential for loops within processes in an unspecified environment, may well prove to be undecidable. We can still only hope to be able to find an approximation even to the anomalies present in the simplified problem.

CHAPTER IV

EARLIER METHODS

In our quest for a satisfactory method for the detection of synchronisation anomalies (and in particular, potential deadlocks) in HAL/S programs, we considered several methods already in existence. These were ultimately rejected for various reasons, but it is useful to discuss a couple of them briefly at this point. This may help to save time for anyone following up on the work presented here. It will serve as a useful introduction to the problems involved in analysing HAL/S programs, even when they are simplified as in chapter three. Finally, other concurrent languages may perform synchronisation in different, more restrictive, ways that bypass these problems, and a variation of one of the methods presented here may be ideally suited to performing analysis on such a languages.

4.1 METHODS INVOLVING RESOURCE DEPENDENCY CYCLES

There have been several papers presenting methods for detecting potential deadlocks involving system resources. One example of such a deadlock would be if process P1 holds resource R1 while attempting to acquire resource R2, and at the same time process P2 holds resource R2 while attempting to acquire resource R1. Clearly neither process can ever progress.

Our early efforts to develop a HAL/S analyser using available methods for the detection of this type of deadlock were based on the paper by Saxena [6]. The basic principle behind his algorithms is to produce a graphical representation of possible resource dependencies. The representation contains a node for each system resource, with a directed edge from one node to another if any process can hold the first resource while attempting to acquire the second. The directed edges are inserted by an examination of each process to determine the possible orders of acquires and releases within that process. Potential deadlocks show up as particular kinds of cycles in the graph. (Note: the above is a considerable simplification of the actual method).

In this system, a process will be stalled only when it attempts to acquire a resource currently held by another process. To apply it to HAL/S, therefore, requires somehow equating waits with acquires, which also involves equating schedules, closes, sets, and resets with acquires and releases, and equating process events and event variables with system resources.

We found that we could satisfactorily produce such an equation to deal with process events and their corresponding operations - schedules, closes, and waits involving process events only. Each process event is simulated by two system resources, one corresponding to the process event having the value true, and one corresponding to it having the value false. A schedule results in the resource corresponding to the process event having the value true being released, and the resource corresponding to the process event having the value false being acquired. A close or return results in the

reverse operations. A wait for (NOT) a process event results in an acquire followed immediately by a release on the resource corresponding to a value of true (false) for the process event. By introducing some extra resources we could satisfactorily treat wait expressions involving ANDs and ORs, and take account of the fact that a process will not start executing until after it has been scheduled.

However, we could not satisfactorily take account of terminates in this simulation, and we found insurmountable problems when we attempted to extend it to take account of event variables and the operations on them. The main problems were as follows:

- i) Event variables cannot conveniently be equated to static resources. For a static resource, the number of releases on that resource must ultimately equal the number of acquires, and there can never have been more releases than acquires. In HAL/S, however, there is no restriction on the relative numbers of sets and resets that can be performed on a particular event variable.
- ii) One of the restrictions for the use of Saxena's algorithms is that a process that acquires a resource must be the one to release that resource. In HAL/S, any process can perform a reset on an event variable after there has been a set on that variable.
- iii) A further restriction with Saxena's algorithms is that, where a loop exists in a process, the resources held by the process at the end of each iteration of the loop must be the same as those held by the process when the loop was first entered. This cannot be applied to event variables in HAL/S.
- iv) Detecting other types of anomalies is not possible.

However, the speed of the algorithms is high and the amount of processing is low, which should make them a definite consideration for other languages.

4.2

PETRI NETS

A Petri net is a model of information flow particularly useful for modeling computer systems having actual or logical concurrency. It is usually represented pictorially in the form of a graph, as here. The graph itself models the static properties of the system, and contains two types of nodes - circles (called places) and bars (called transitions). These places and transitions are connected by directed arcs, each arc going either from a place to a transition, or from a transition to a place.

In addition to the static properties resulting from the topology of the graph, Petri nets have dynamic properties resulting from 'executing' the graph. For this purpose, the Petri net has a number of markers (called tokens) which reside in places, and which are created and destroyed during the execution of the net according to a fixed set of rules. These rules are as follows:

- i) If all the input places at a transition (i.e. places having an arc going into the transition) have at least one token, that transition is 'enabled', and can fire.
- ii) When a transition fires, one token is removed from each input place, and one token is added to each output place (i.e. a place with an arc coming from the transition).
- iii) The firing of a transition is instantaneous; i.e. there is an order to the firings, and no two firings can occur simultaneously.

iv) If, at any time, more than one transition is enabled, an arbitrary choice is made as to which of the enabled transitions fires.

v) The Petri net will usually have an initial configuration of markers. An execution of the net consists of repeatedly firing enabled transitions until there are no remaining enabled transitions.

The above represents a considerable simplification. For more information see e.g. Peterson [7].

We found that we could conveniently simulate all the synchronisation statements in HAL/S with Petri net structures.

Each process has a structure somewhat resembling a flowgraph, with a transition (or transitions) for each statement, and a place before each statement representing flow of control. At any given time, a process which is on the process queue has a token in exactly one of its flow of control places corresponding to the current position of the flow of control. A process which is not on the process queue has no such token. A transition is enabled when a token appears in the place before it, and when a transition fires (representing the execution of a statement) a token appears in the place before the transition representing the next statement that would be executed.

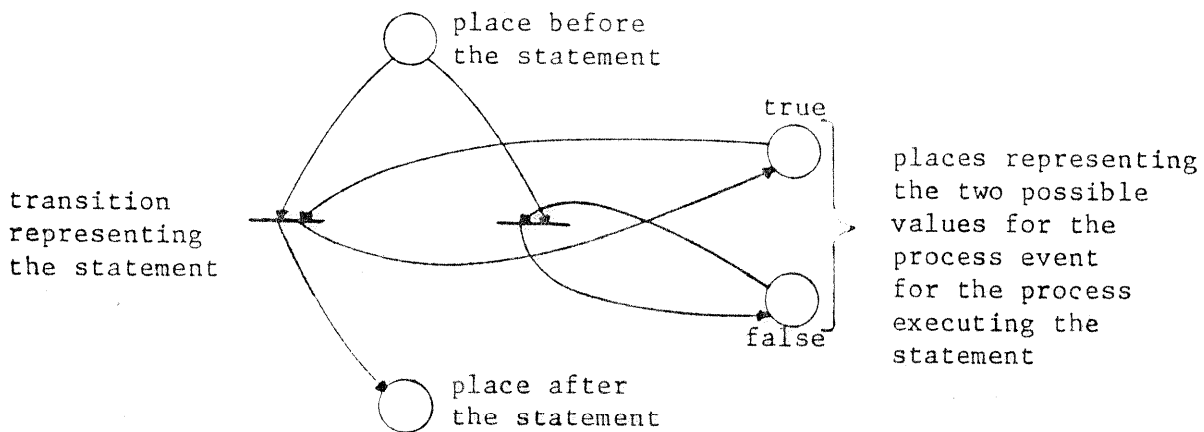
Each process event and event variable is represented by two places, corresponding to the two possible values (true and false) for the variable. At any given time, there is a token in one of the two places for each variable giving the current value of the variable. Initially tokens are inserted in the place representing a value of true for the process event of the main program, in the place

representing a value of false for all other process events, and in one of the two places for each event variable representing the initial value of that event variable.

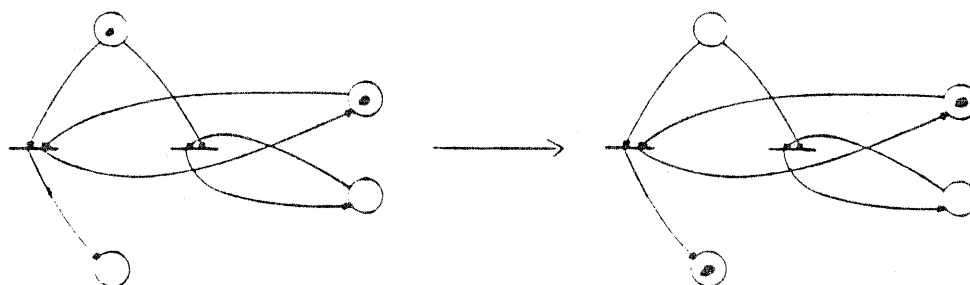
The termination of a process results in the token for the corresponding process event being moved to the place representing a value of false. Should this occur before the normal completion of the process, the process does not execute any more instructions and is immediately removed from the process queue. To account for this, each transition (or transitions) representing a statement in the process can only fire as long as there is a token in the place representing a value of true for the process event of that process. Otherwise the flow of control token is removed. An example of this appears in figure 1. Such a construct is present at each statement, but has been left out of figures 2-12 for ease of understanding. The termination of a process is thus simulated by moving the marker representing the value of the process event from the place representing the value true to the place representing the value false. (see figure 2).

A branch in a process is represented by a branch construct, as in figure 3. A join of paths is simply represented by having more than one arc coming into the place before the statement where the join occurs. A loop is represented by a branch and join, the exact form of which depends on whether the loop must be executed at least once or not (see figures 4 and 5).

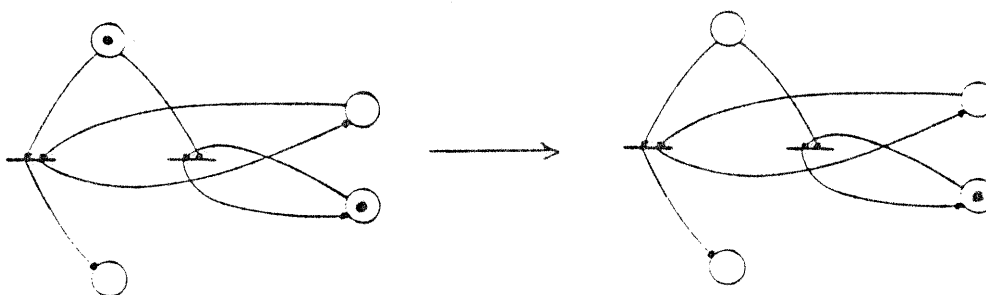
The different synchronisation statements each have different constructs, as shown in figures 6-12. For various of these statements, there are places in the constructs representing errors in



PETRI NET MODEL OF THE STRUCTURE THAT APPEARS AT EACH HAL/S
STATEMENT TO TAKE ACCOUNT OF PREMATURE TERMINATIONS

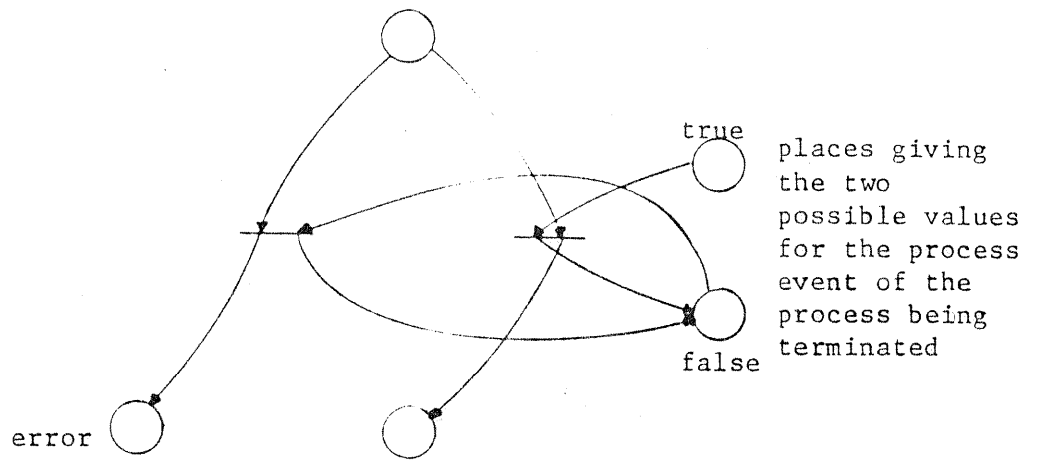


A. Process has not been terminated, and statement is executed as normal.

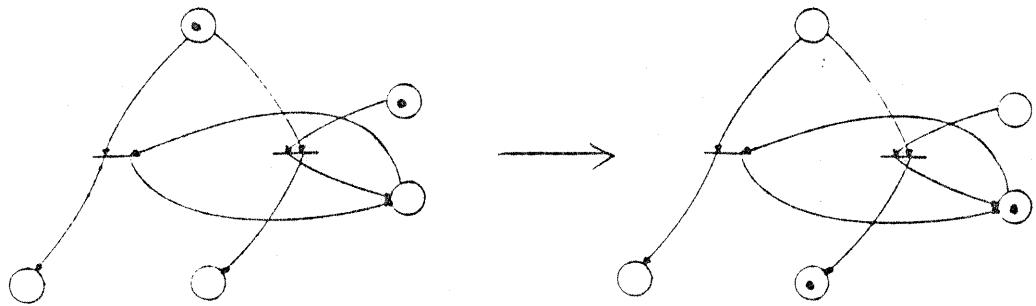


B. Process has been terminated, so flow-of-control marker is removed.

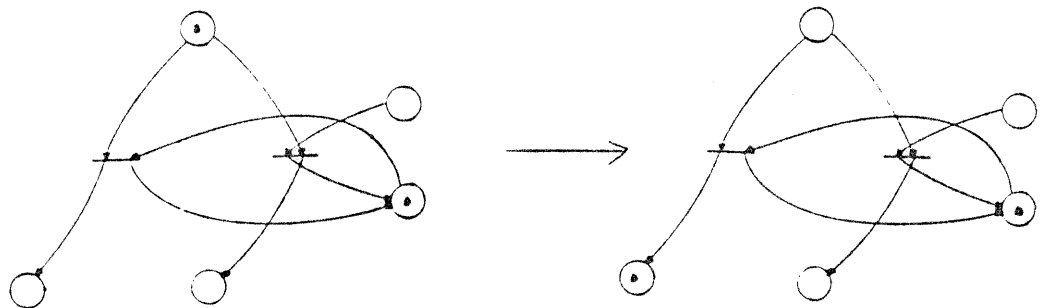
Figure 1.



PETRI NET MODEL OF THE TERMINATE STATEMENT.

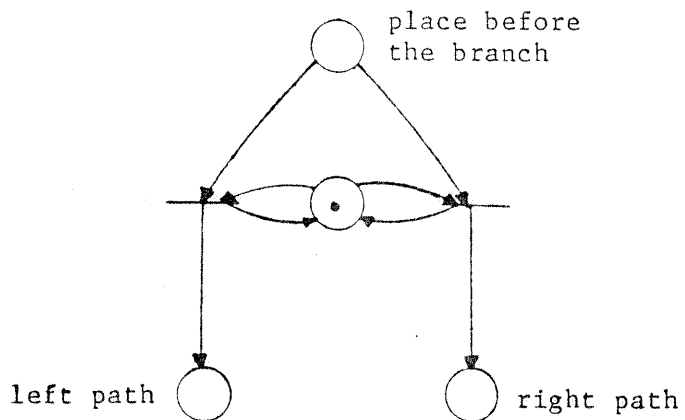


A. Process being terminated is active.

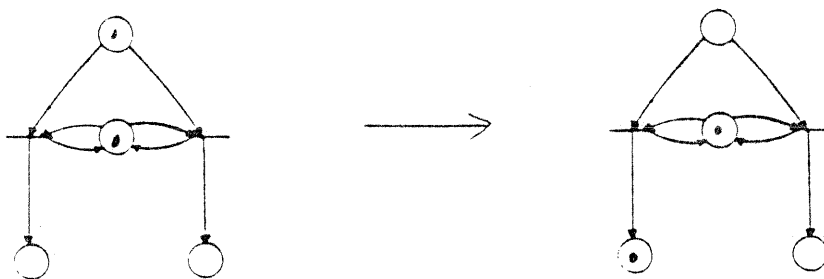


B. Process being terminated is not active.

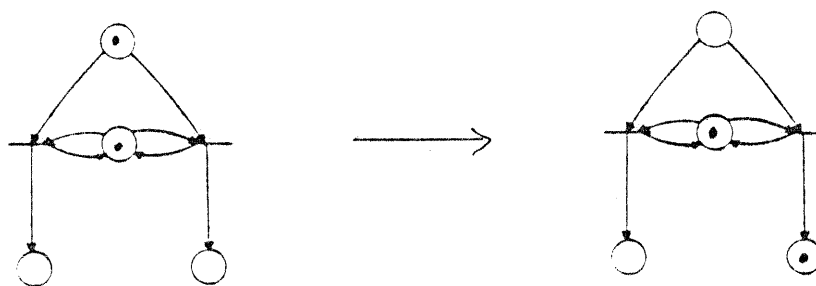
Figure 2.



PETRI NET MODEL OF A TWO WAY BRANCH.

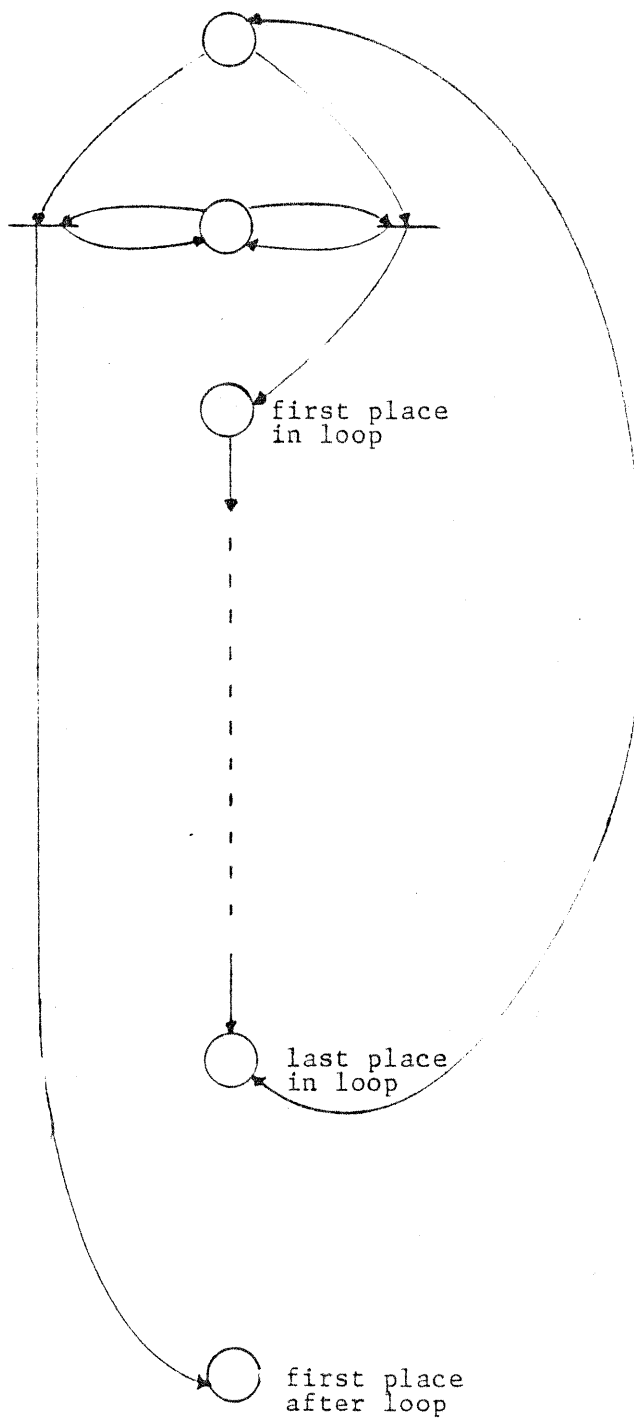


A. Left hand path is taken.



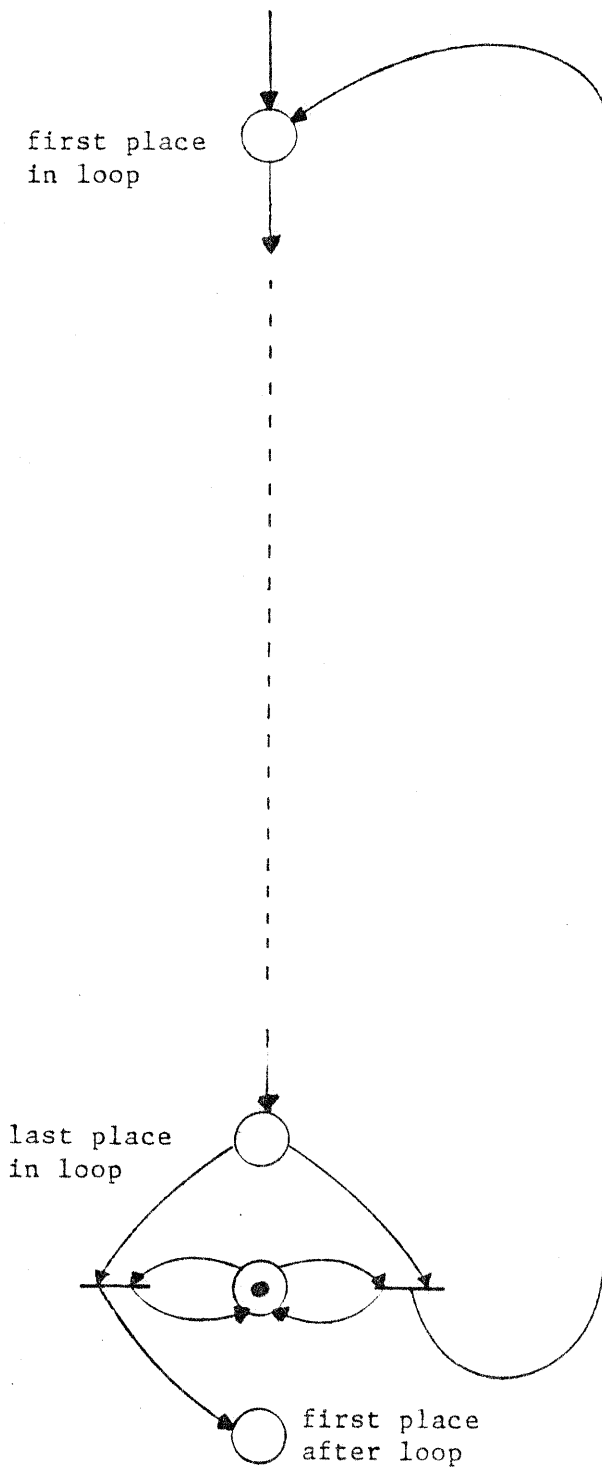
B. Right hand path is taken.

Figure 3.



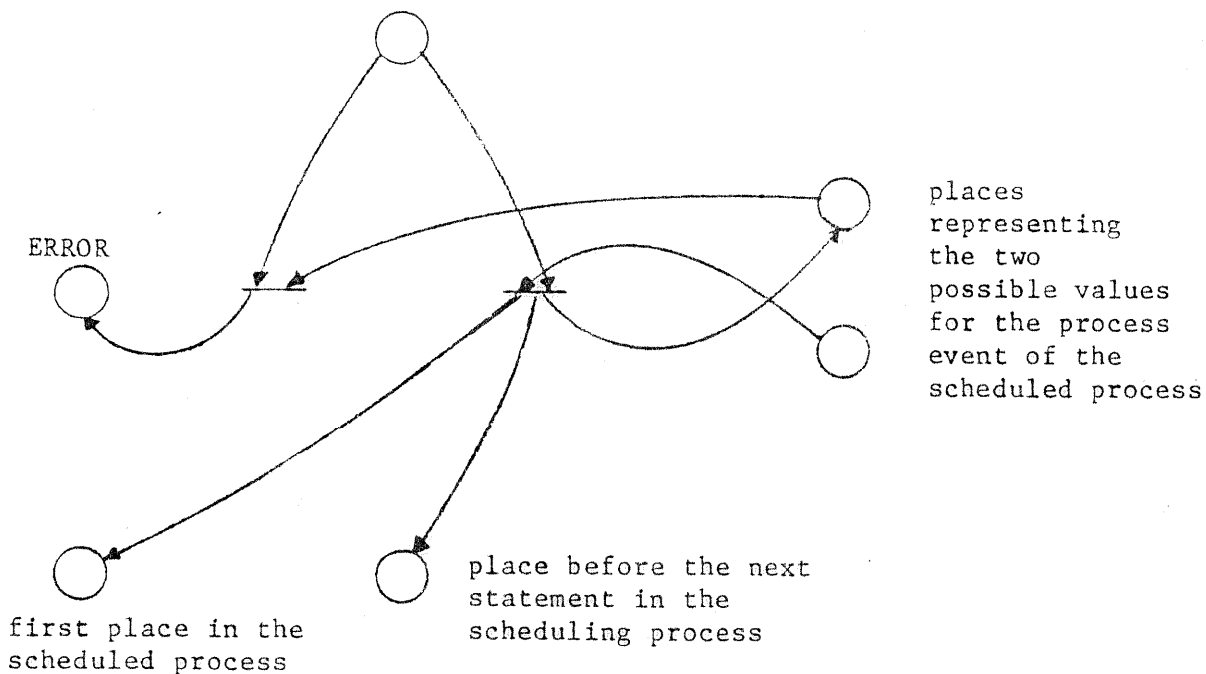
PETRI NET MODEL OF A LOOP THAT NEED NOT BE EXECUTED EVEN ONCE

Figure 4.

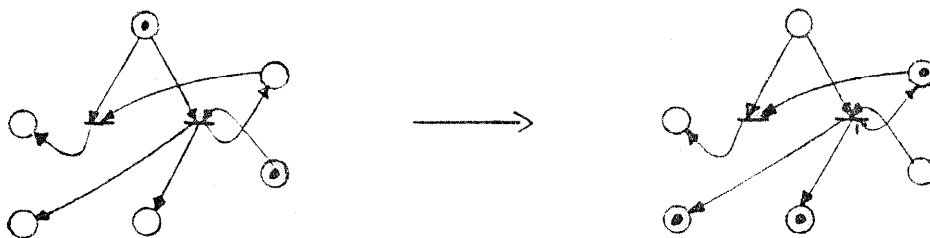


PETRI NET MODEL OF A LOOP THAT MUST BE EXECUTED AT LEAST ONCE

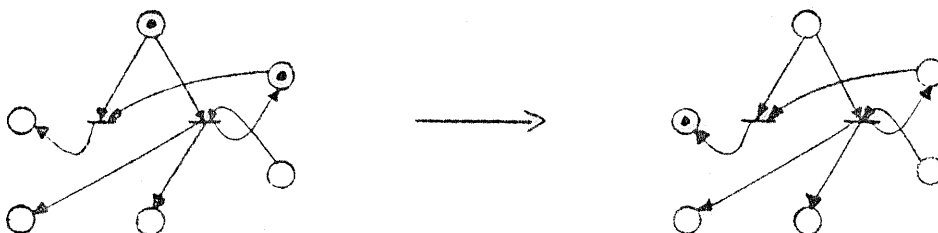
figure 5.



PETRI NET MODEL OF THE SCHEDULE STATEMENT

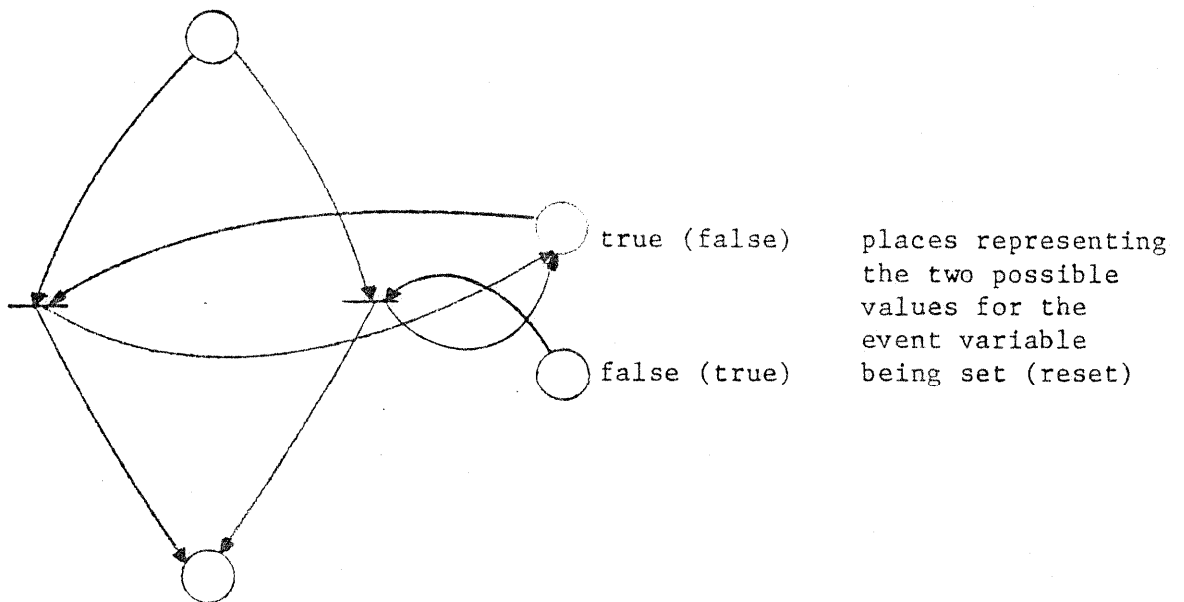


A. The scheduled process was inactive at the time of scheduling

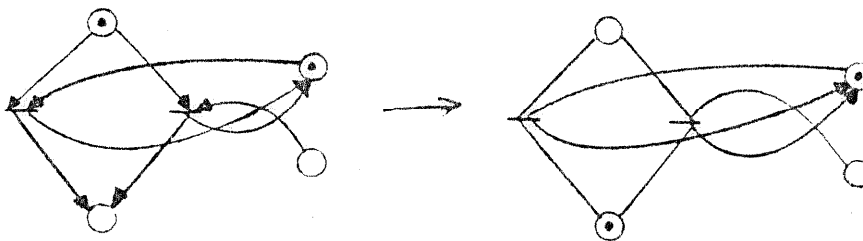


B. The scheduled process was already active at the time of scheduling

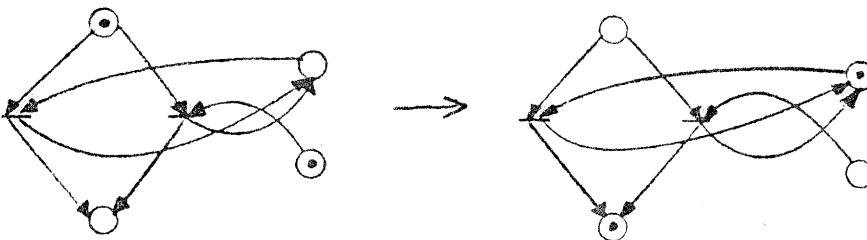
Figure 6 .



PETRI NET MODEL OF THE SET (RESET) STATEMENT.

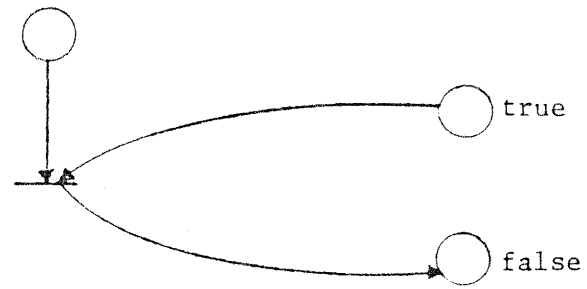


A. Event variable was 'true' ('false') prior to the set (reset).

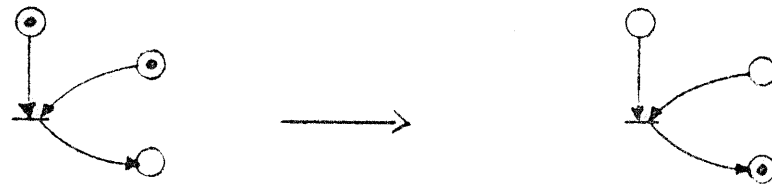


B. Event variable was 'false' ('true') prior to the set (reset).

Figure 7.

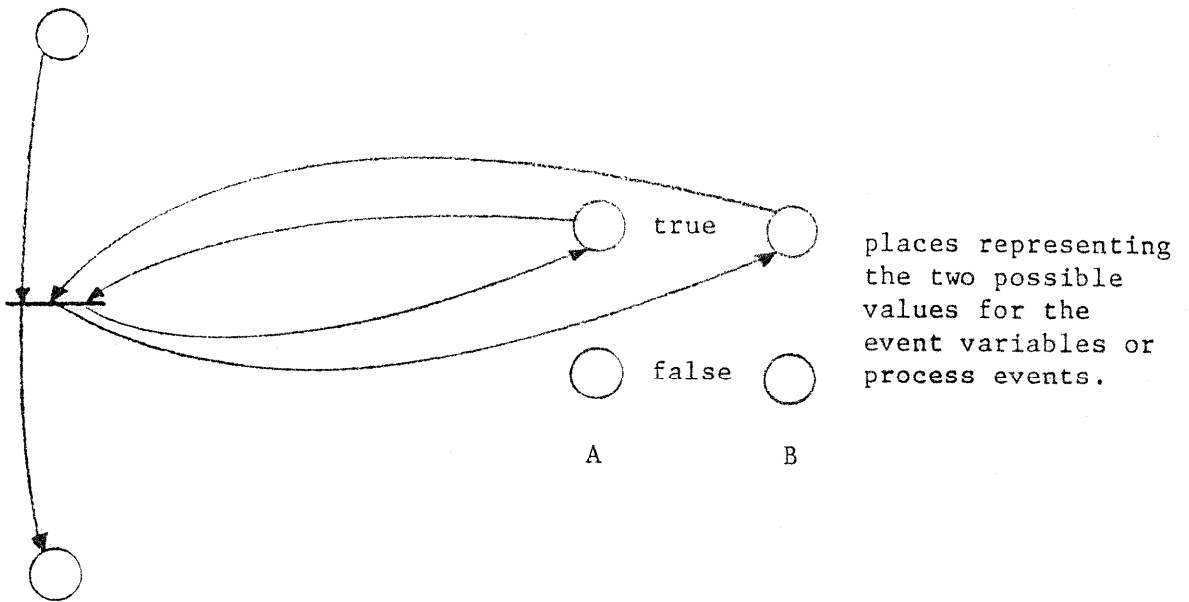


places representing
the two possible
values for the
process event of
the closing
process.



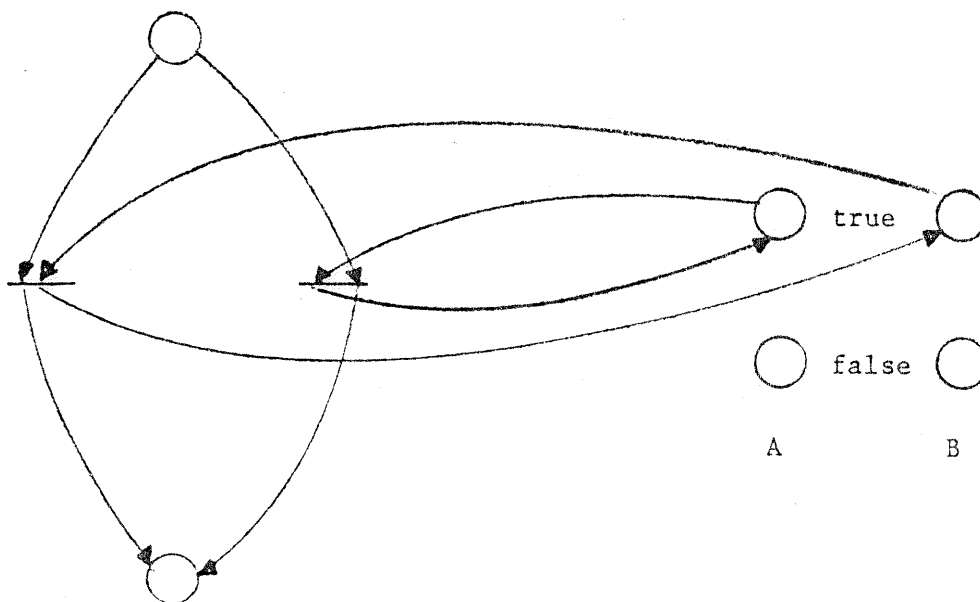
PETRI NET MODEL OF THE CLOSE STATEMENT

Figure 8.



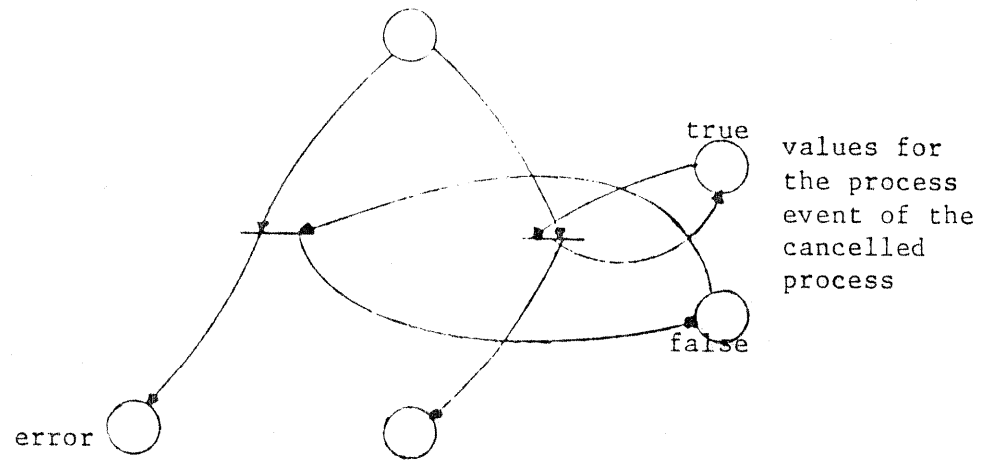
PETRI NET MODEL OF THE STATEMENT 'WAIT FOR A AND B'

Figure 9.



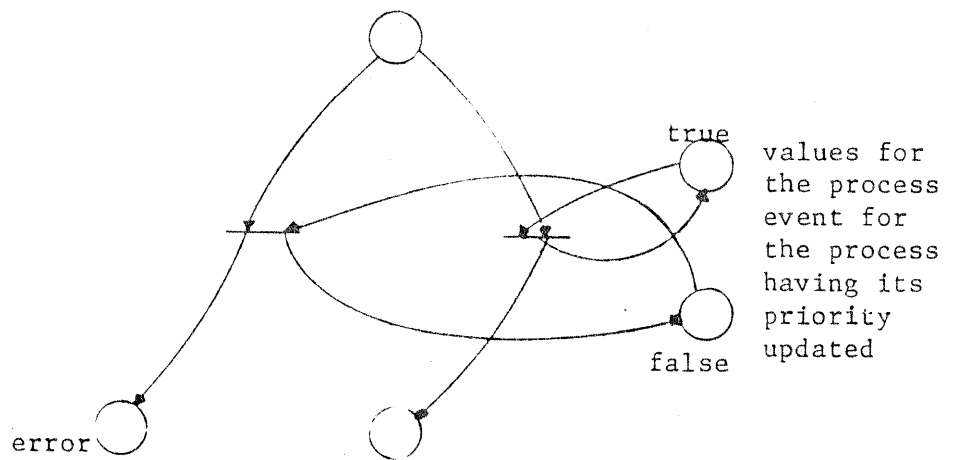
PETRI NET MODEL OF THE STATEMENT 'WAIT FOR A OR B'

Figure 10.



PETRI NET MODEL OF THE CANCEL STATEMENT

Figure 11.



PETRI NET MODEL OF THE UPDATE PRIORITY STATEMENT

Figure 12.

the program. When a token appears in such a place, this signifies that a run-time error would have occurred in the HAL/S program. For anomalies other than deadlock, therefore, determining whether the program contains such anomalies reduces to determining whether the corresponding error places are reachable. Deadlock is more complicated.

One important point to note about the Petri net model of a HAL/S program is that such a net is 1-bounded; i.e. never more than one token can reside in a single place at a time. This is important from complexity considerations. Initially a token is placed in the first flow of control place for the main program, corresponding to the main program being placed on the process queue by some means other than the execution of a schedule statement.

The reasons why this line of investigation was subsequently rejected are two-fold. Firstly, the algorithms for detecting the anomalies we are interested in are at least exponential in their time-bounds. (Reachability for a 1-bounded net is exponential, and deadlock may be even worse). Secondly, to the best of our knowledge, efficient approximation algorithms are not currently available for deadlock and reachability, which would guarantee to find a small superset of all such instances.

However, analysis of HAL/S programs by modeling with Petri nets should provide useful information when more is known about the properties of Petri nets. This line of investigation may prove useful in the future.

CHAPTER V

THE INTER PROCESS PRECEDENCE GRAPH

The system that we finally developed for anomaly detection uses a graphical representation of HAL/S programs that seems more amenable to analysis than the Petri net representation. Our representation, which we call an Inter Process Precedence Graph (IPPG), contains information about the possible paths through each process, in addition to information about forced and possible orderings of the individual statements in different processes.

5.1 FURTHER SIMPLIFICATION OF THE PROBLEM

The system performs its analysis on a further simplification of HAL/S containing only five types of synchronisation statements. These are SCHEDULE (with no conditions on the execution of the scheduled process), CLOSE (without the need to wait for the completion of dependent processes), SET, RESET, and WAIT. In addition, it allows for statements which cause run-time errors if executed at the wrong time, but which otherwise do not affect the analysis. These are CANCEL, UPDATE PRIORITY, and TERMINATE (see viii below).

We have therefore produced approximate encodings for the other synchronisation constructs and facilities, which preserve the anomalies in the program under analysis, and which utilise only branches and the five remaining types of statements. Note that in

order to prove that the anomalies in the original program are preserved by the encoding, it is only necessary to prove that the encoding allows for execution sequences that have the same effect as the execution sequences in the unencoded program.

i) Return.

A return statement in a process is treated as a branch to the close statement. Clearly, among the possible resulting execution sequences are all those where the close is executed immediately following the execution of the return statement, as actually occurs when the program is executed. Thus anomalies are preserved.

ii) Signal on an unlatched event variable.

This is encoded as a set followed immediately by a reset on an latched event variable having the same name as the unlatched event variable. Clearly, among the possible execution sequences that result from this are all those where the reset occurs immediately following the completion of any outstanding waits for a value of true for the event variable, as would actually occur during normal execution of the signal statement. Thus anomalies are preserved in this encoding. Note that a wait for a value of false for the event variable is redundant, since the value will be false at all times except for the (infinitesimally short) time that the variable will be true during the execution of a signal statement on it. Accordingly, any such terms that appear in a wait expression are ignored.

iii) Signal on a latched event variable.

This is also encoded using a set and a reset, with the order of the two operations depending upon the current value of the event variable. The encoding of 'SIGNAL EV' is therefore as follows:

```

IF EV
THEN DO
    RESET EV;
    SET EV;
END;
ELSE DO
    SET EV;
    RESET EV;
END;

```

Among the possible execution sequences are all those where the correct path is taken. The discussion for ii above applies here also, to imply that on those execution sequences where the correct path is taken, anomalies are preserved.

To avoid the addition of anomalies not present in the actual HAL/S program under analysis, an additional step is taken for signal statements during the construction of the inter process precedence graph (see 5.4 and 8.1).

iv) Scheduling a process to immediately enter the stalled state.

Where such a situation occurs, with the criterion for the removal of the scheduled process from the stalled state being the satisfying of a specified event expression, the scheduled process has a wait for that event expression inserted as its first statement. Among the possible resulting execution sequences are all those where the wait is reached, and hence the process is stalled, immediately following the execution of the schedule statement, as would actually occur during execution of the program. Therefore, anomalies are preserved.

v) Scheduling a process to execute cyclically.

Where a process can be scheduled to execute cyclically, that process has a conditional branch inserted immediately before its close statement, going to the first statement in the process (excluding any wait inserted for iv above). With the insertion of this branch, the body of the process can be executed any arbitrary number of times following the completion of any initial stall condition, and prior to the removal of the process from the process queue. This is exactly what occurs during execution of the program, so anomalies are preserved. An important point to note is that the final graphical representation of the HAL/S program contains a different representation of a process for each instance of that process being scheduled. Each such representation is individually tailored to reflect the particular scheduling conditions for the instance of the process that it represents. This also applies for iv above.

vi) A process scheduled with a terminating criterion.

In such situations we assume that the termination criterion may be satisfied at the time of scheduling, and hence that the process may not be executed at all. We therefore insert a conditional branch around the schedule statement in the scheduling process, to reflect that the schedule statement may have no effect at all. Clearly, this will preserve anomalies.

vii) Close.

The closing of a process involves two distinct operations, namely waiting for the completion of any dependent sons still on the process queue, followed by the removal of the closing process itself

from the process queue. A close statement is therefore encoded as a wait for the completion of all dependents followed by a simple close involving no dependents. Among the possible execution sequences are all those where the simple close is reached, and hence the process is removed from the process queue, immediately following the completion of the last remaining dependent son. This is exactly what happens during execution of the program, and so anomalies are preserved.

viii) Terminate.

There are two aspects to a terminate statement that are taken into consideration by the analysis system. The first of these is that a process which is prematurely terminated is immediately removed from the process queue, along with all dependent sons, grandsons, etc. These terminated processes do not complete execution, and hence contain statements which are normally executed, but in this case are not executed. Accordingly, during the building of the IPPG, wherever a terminate statement is encountered, all processes that can be terminated by that statement are considered, and the earliest point in the execution of each such process at which the termination could occur is determined. From this point onwards within the process, a conditional branch is inserted before each statement in the process going to the close statement. This indicates that the process may be terminated at any time following this earliest point, and that at such a time the process is immediately removed from the process queue without executing any further statements. Among the resulting execution sequences are all those where each process which is prematurely terminated reaches an arbitrary point in its execution following the earliest point at which the termination can occur,

branches to the close statement, and is immediately removed from the process queue. This simulates exactly what happens during execution of the terminate statement, and so anomalies are preserved.

The second aspect that the analysis takes account of is the time at which the terminate statement can execute. If the process(es) being terminated has(have) already completed execution, this represents a run-time error, and hence an anomaly. If not, this represents a premature termination, which is also an anomaly. Accordingly, the terminate statement is left in the process but its only effect on the analysis is in the determination of such anomalies.

5.2

PROCESS FLOW GRAPHS

The information about possible execution paths through each of the processes is stored graphically in the form of process flowgraphs, one flowgraph for each process. A flowgraph is a directed graph, whose nodes represent statements in the process, and whose edges (called Flow Of Control Edges, or FOCEs) represent direct flow of control paths from one statement to another.

Each process flowgraph has a single entry node (a node with no incoming FOCEs) and a single exit node (a node with no outgoing FOCEs). The entry node represents the first action the process performs when scheduled, and the exit node represents the last action performed before completion. Since return statements are encoded as branches to the close statement, the only exit from the process can be through the execution of the close statement, and hence this is the exit node. In a process flowgraph, there are paths from the

entry node to all other nodes, and paths to the exit node from all other nodes (unless the process contains an infinite loop).

The flowgraph for a particular process is derived from the parse tree and symbol table for that process. It is then simplified to contain only those nodes representing synchronisation statements in the simplified encoding of that process*. An edge from one node to another indicates that a flow of control path exists in which the latter node is the first synchronisation statement in the process to be executed after the former node. The information about how a process is scheduled is not available when its flowgraph is built, so insertion of any initial wait node, and any conditional branch indicating that the process was scheduled cyclically, is left until later (see 5.4).

The building of the original flowgraph for a process follows standard procedures and is not discussed here. For details see e.g. [5] and [8]. The reduction of the flowgraph to contain only synchronisation statements uses a breadth-first search. At the same time as this is happening, subroutines are inserted in-line, and loops are transformed (see 5.3). The algorithms for this appear in appendix B.1.

5.3

LOOPS

Loops in a process appear as strongly connected components in the process's flowgraph. Loops containing synchronisation statements

* Note: From now on, the terms 'node' and 'statement' are used more or less interchangeably, avoiding repetition of phrases such as "the node representing statement...." and "the statement represented by....".

do currently present problems for our analysis. The problems stem from the fact that situations can exist in which an error will only occur if a particular loop (or set of loops) executes a particular number (or combination of numbers) of times. To guarantee to find all potential errors, therefore, requires checking all possible combinations of cycles around all the loops in all the processes.

Currently, we do not have a satisfactory method for doing this. Instead, we compromise by setting a limit on the maximum number of cycles around each loop, and we check for anomalies in all possible combinations of cycles (up to this maximum) around all the loops in all the processes.

The time and space requirements of the analysis increase rapidly as this limit increases. Accordingly, we have set this limit at two. We maintain that even given this low limit we shall still find by far the majority of anomalies in a 'normal' program. For instance, we will find all anomalies resulting from a particular loop not executing at all, or from a particular loop executing less than, more than, or an equal number of times as any other particular loop. At the same time, we do hope to be able to rectify this problem (see 9.1).

It has also been necessary to place a restriction on the use of loops containing synchronisation statements. In terms of the process flowgraphs, the restriction is that each strongly connected component can only have a single entry node (a node with incoming FOCEs from nodes outside the strongly connected component). In terms of the HAL/S program, this implies that every loop containing synchronisation statements must be entered at the top (i.e. it is

not allowable to jump into a loop), and each path at a conditional statement can only be entered via the conditional statement (i.e. it is not allowable to jump into the THEN or ELSE path at a branch). However, good programming practice normally dictates that this is the case anyway.

5.4

INTER PROCESS PRECEDENCE EDGES

The process flowgraphs contain information about the orders of execution of the synchronisation statements within processes. The information about possible and forced orders of execution among the individual statements in different processes is contained in the Inter Process Precedence Graph (IPPG). This consists of one or more copies of each process flowgraph, along with a number of Inter Process Precedence edges (IPPEs - represented pictorially as wavy lines) each of which connects a pair of nodes in different flowgraphs.

The presence of an IPPE in the IPPG should indicate that in at least one execution sequence it is the execution of the predecessor node of the IPPE that allows for the process containing the successor node to enter the ready state from having been in some other state. There are two different situations in which this occurs:

The execution of a schedule statement places the scheduled process in the ready state on the process queue. The process was not previously on the process queue. Hence there is an IPPE going from each schedule node to the first node in a copy of the scheduled process's flowgraph. There is a new copy of that flowgraph for each different schedule on that process in the IPPG. The IPPG remains

finite and bounded in size, however, since recursion of any sort is not possible in HAL/S.

The execution of a set, reset, schedule, or close statement may satisfy a particular wait expression, thus placing a process in the ready state from having been in the stalled state. Hence there is an IPPE going from each set, reset, schedule, and close to each wait node whose wait expression can be satisfied by the execution of the set, reset, schedule, or close.

The building of the IPPG is a four stage process. The first stage consists of inserting all the required copies of process flowgraphs, along with a pair of IPPEs for each copy (except the main program) so inserted. The first IPPE of each pair goes from the schedule node in the scheduling process to the entry node of the scheduled process. The second goes from the exit node of the scheduled process to either the wait for dependents before the close of the scheduling process (if the scheduled process is a dependent son) or to the wait for dependents before the close of the enclosing program (otherwise). The two IPPEs effectively determine the maximum amount of time, relative to all other processes, that this instantiation of the process can be on the process queue.

The algorithm for this first stage consists of initially inserting the flowgraph for the main program into the IPPG. This is inspected, and wherever a schedule exists, a copy of the scheduled process's flowgraph is inserted, along with its pair of IPPEs. As each copy of a process flowgraph is inserted, that copy is recursively inspected for any schedules that it may contain, and copies of these scheduled process's flowgraphs are also inserted etc.

The second stage consists of inserting all IPPEs that could possibly be required. Each wait node in the graph is considered in turn. The wait expression is first translated in conjunctive normal form. Subsequently, each term in the expression is considered in turn. IPPEs are added going to the wait node from all nodes in the graph whose execution would set that term to true, and also from the first node of the main program if the term is true initially. If the term is for a process event to be true, the nodes whose execution set the term to true are all schedules on that process. Likewise, the nodes setting a process event to false are all close nodes for copies of that process. For an event variable, the nodes setting it to true are all sets on the variable, and those setting it to false are all resets on the variable. There is one exception to this rule however, concerning signal statements on unlatched event variables. Where there is a wait for a value of true for a latched event variable, and there is a signal on that variable, only one IPPE is added coming from the signal although in the encoding of a signal there are actually two sets on the variable (see 5.1). The IPPE is added from the set that appears on the path that is taken if the event variable were false when the signal is executed, since if the event variable were true the signal would have no effect on the completion of the wait. Similarly, if there is a wait for a value of false for the event variable, the only IPPE added from the signal construct is from the reset on the path that is taken if the event variable is true when the signal is executed.

The incoming IPPEs at a wait node are grouped into conjunctive normal form such that the predecessor node of only one IPPE in a conjunct group has to execute for the corresponding conjunct in the wait expression to be true.

The third and fourth stages of building the IPPG are discussed in sections 6.3 and 6.4.

The algorithms for building the IPPG appear in appendix B.2.

CHAPTER VI

EXECUTION SEQUENCE SETS

The inter process precedence graph is used in the generation of a number of sets of nodes, collectively known as the execution sequence sets, at each node in the graph. These execution sequence sets give information about the possible and forced orders of execution of the nodes in the graph.

6.1

EXECUTION SEQUENCE SETS

Before we discuss the contents of the individual sets, we first explain how we use the term 'execution path' in relation to concurrent programs.

An execution path can be regarded as the set of all possible execution sequences for the particular set of statements that are executed during an execution of the program. In other words, given that a particular set of statements will be executed in one particular run of the program, the execution path for that set of statements contains all possible sequences in which those statements could be executed. It is therefore a set of paths through each individual process, together with the partial orderings that are enforced by the synchronisation statements in those paths. For a program with no potential concurrency, the set of execution sequences in an execution path contains only one element, i.e. the two terms refer to the same thing. For a program with concurrency, an

execution path contains no information concerning the actual order of execution of the individual statements in sections of two or more processes running in parallel.

At a node n , the following execution sequence sets are generated:

ALWAYS(N) contains all nodes which must execute if N is to execute; i.e. those nodes that are always present in every execution sequence containing N .

NEVER(N) contains all nodes which cannot execute if N is to execute; i.e. those nodes that do not appear in any execution sequences containing N .

BEFORE(N) contains all nodes which, if they execute at all, will execute before N ; i.e. those nodes that appear before N in at least one execution sequence containing N , but do not appear after N in any execution sequences containing N .

ALWAYS BEFORE(N) is the subset of BEFORE(N) that contains those nodes that always execute before N in all execution sequences containing N .

POSSIBLY BEFORE(N) is the superset of BEFORE(N) containing those nodes which, for at least one execution path containing N , will execute before N in all execution sequences in that execution path.

AFTER(N) contains those nodes which, if they execute at all, will execute after N ; i.e. those nodes that occur after N in at least one execution sequence containing N , but do not occur before N in any execution sequences containing N .

ALWAYS AFTER(N) is the subset of AFTER(N) containing those nodes that always occur after N in all execution sequences containing N.

POSSIBLY AFTER(N) is the superset of AFTER(N) containing those nodes which, for at least one execution path containing N, will execute after N in all execution sequences in that execution path.

There is a certain amount of symmetry implied by the above definitions of the execution sequence sets. Firstly if node A is in the NEVER set of node B then node B will be in the NEVER set of node A. If node A is in the BEFORE set of node B then node B will be in the AFTER set of node A, and vice versa. Similarly, if node A is in the POSSIBLY_BEFORE set of node B then node B will be in the POSSIBLY_AFTER set of node A, and vice versa.

6.2 GENERATION OF THE EXECUTION SEQUENCE SETS

To generate the execution sequence sets, it is first necessary to generate some intermediate sets. BEFORE_WITHIN_PROCESS(N) contains those nodes within the same process as N which belong to BEFORE(N). Similarly for AFTER_WITHIN_PROCESS(N). ALWAYS_BEFORE_WITHIN_PROCESS(N) contains those nodes within the same process as N that belong to ALWAYS_BEFORE(N). Similarly for ALWAYS_AFTER_WITHIN_PROCESS(N). The algorithms to generate these sets use a breadth-first search, and take advantage of the fact that the process flowgraphs do not contain any loops at this stage.

From these sets are generated the ALWAYS_SUBPROCESS and SUBPROCESS sets for each process. For a process P, SUBPROCESS(P) contains all processes which are scheduled from within P, or from

within a process itself scheduled from within P, etc. ALWAYS_SUBPROCESS(P) is the subset of SUBPROCESS(P) containing those processes scheduled (possibly indirectly) by P which always execute if P executes.

The next step in the generation of the ALWAYS sets is the generation of the ALWAYS set for the first node of the main program. This contains those nodes that must always execute, within the main program and within any other processes, whenever the program is run. For all other processes, the ALWAYS set of the entry node is equal to the ALWAYS set of the schedule node for that process.

For a general node N, ALWAYS(N) contains the following nodes:

- i) The ALWAYS set of the entry node of the process containing N;
- ii) Other nodes within the same process as N, but not contained in i above, that must always execute if N executes;
- iii) If ii above contains any schedules, then any nodes in those scheduled processes, and processes in their ALWAYS_SUBPROCESS sets, which must always execute if the process containing them executes.

To generate the NEVER sets, it is first necessary to generate NEVER_WITHIN_PROCESS sets. NEVER_WITHIN_PROCESS(N) contains those nodes within the same process as N that cannot execute if N executes. It consists of those nodes within the same process as N not contained in BEFORE_WITHIN_PROCESS(N) or AFTER_WITHIN_PROCESS(N).

For each process, except the main program, the NEVER set of the entry node is equal to the NEVER set of the schedule node for that process. For a general node N, NEVER(N) contains the following nodes:

- i) the NEVER set of the entry node of the process containing N;
- ii) NEVER_WITHIN_PROCESS(N);
- iii) If ii contains any schedules, then all nodes in those scheduled processes, and all nodes in any processes belonging to the SUBPROCESS sets of those processes.

CALCULATE_BEFORE(N,P) is a recursive function which returns as its value the BEFORE set of a node N in a process P. Nodes in BEFORE_WITHIN_PROCESS(N) are members of BEFORE(N). Nodes in the intersection of the BEFORE and NEVER sets of nodes within P that have edges to N are placed in BEFORE(N) as well as nodes within other processes which are the tails of IPPEs in a single conjunct group going to N.

Because of the possibility of loops in the graph involving IPPEs, a stack, SAVE, is kept containing an entry (N,P) each time CALCULATE_BEFORE is called. In the event that the recursive process leads to the determination of the BEFORE set of a node N which occurs within the same process as a node M on SAVE, and N is a descendent of M or an element of NEVER(M), then the identity is returned as the result of the call. Also, all entries on SAVE from the one after (M,P) to the top are placed in REDO, indicating that their BEFORE set calculations are valid only from the standpoint of node M, and their true BEFORE sets need to be calculated separately.

CALCULATE_AFTER(N,P) employs the BEFORE sets, reasoning that if M belongs to BEFORE(N) then N belongs to AFTER(M).

The POSSIBLY_BEFORE set of each node N contains each entry node E to N, plus BEFORE(E). The POSSIBLY_AFTER sets are generated from the POSSIBLY_BEFORE sets in the same manner as above.

The algorithms for execution sequence set generation appear in appendix B.3.

6.3

SPURIOUS IPPE ELIMINATION

The third step in the construction of the inter process precedence graph is the removal of IPPEs which reflect impossible execution sequences.

The presence of each IPPE in the graph should indicate that three conditions have been satisfied:

- i) The predecessor node causes a term in the wait expression of the successor node to become true.
- ii) The predecessor node will execute before the successor node in at least one execution sequence.
- iii) In at least one of those execution sequences in ii above, the term does not become false again before the wait has completed.

During the second stage of the graph building, however, all IPPEs are inserted that satisfy only condition i above, and it is possible for some of these to violate conditions ii or iii above. Those that do are spurious, and for more accurate results should be removed prior to performing any analysis.

Figure 13a contains an example of spurious IPPEs. It shows a section of an inter process precedence graph as it would appear immediately following IPPE insertion. The section corresponds to parts of two parallel processes synchronising themselves using one event variable, *ev*. Originally, *ev* has the value false, and no other processes use it. The node numbering is chosen arbitrarily. The presence of an IPPE from node 3 to node 4 should indicate that in at

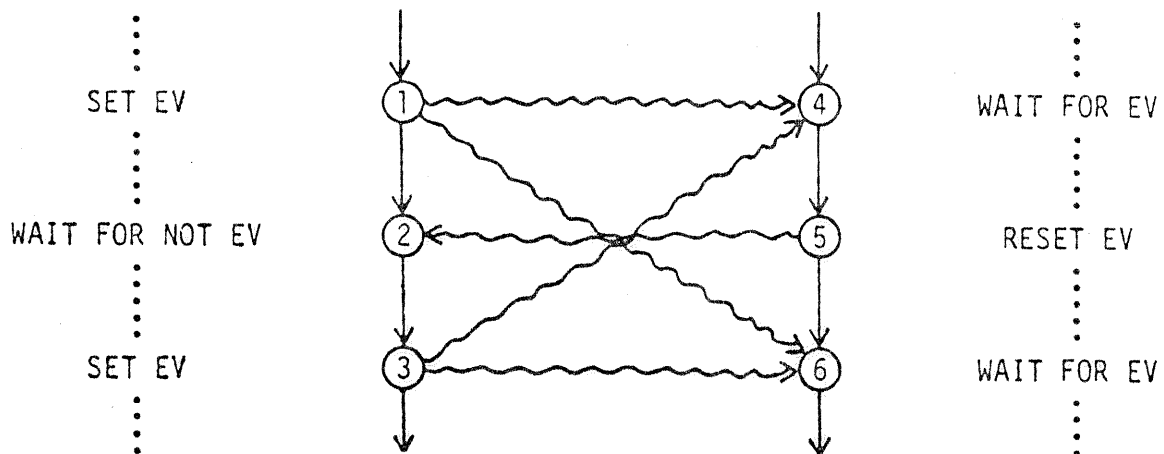


Figure 13a

A section of an inter process precedence graph prior to the removal of spurious IPPEs.

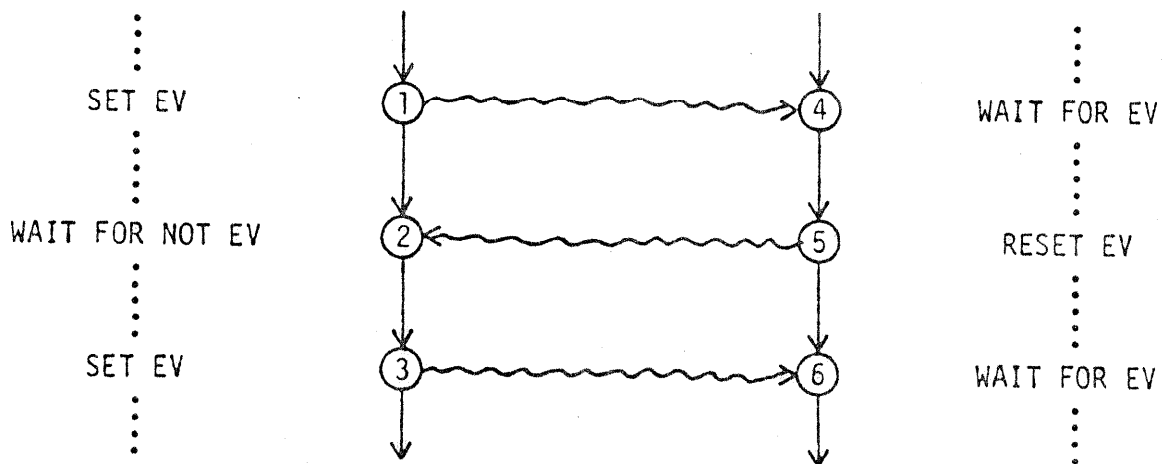


Figure 13b

The same section as it would appear following spurious IPPE elimination.

least one execution sequence, it is the execution of node 3 that allows for the completion of the wait at node 4. However, inspection of the code reveals that node 5 must execute before the wait at node 2 can complete, preventing node 3 from executing until after the wait at node 4 has completed. The IPPE therefore violates condition ii, and should be removed. In addition, the IPPE from node 1 to node 6 should indicate that the wait at node 6 can be completed as soon as node 1 executes. However, node 1 must always execute before the wait at node 4 can complete, and hence its effect will always be negated by the execution of node 5 before node 6 can be reached. This IPPE violates condition iii, and should be removed. Figure 13b contains the section of the inter process precedence graph as it should appear, and inspection of the code will reveal that the execution ordering enforced by the remaining IPPEs is genuine.

Spurious IPPEs can be removed by a consideration of the execution sequence sets for the nodes in the graph. If, for any IPPE, the predecessor node is in the AFTER or NEVER sets of the successor node, condition ii is violated and the IPPE is removed. If, for any IPPE, a node negating the effect of the predecessor node occurs in the intersection of the ALWAYS_AFTER set of the predecessor node and the BEFORE set of the successor node, the IPPE violates condition iii and is removed. The removal of an IPPE may alter the generated execution sequence sets, and so these must be regenerated after an IPPE is removed. The process iterates until no more spurious IPPEs can be found.

If, at any time during spurious IPPE elimination, a node is found to be in its own BEFORE or AFTER set, this indicates the presence of a guaranteed deadlock in the code. The effect of the deadlock may permeate throughout the entire graph in an unpredictable manner, potentially causing much 'cascading' of errors.

6.4

TAKING TERMINATES INTO ACCOUNT

The final stage in the construction of the graph is the revision to the graph that is necessary to take account of any potential premature terminations. As specified in section 5.1, such revisions take the form of the addition of conditional branches into processes that can potentially be prematurely terminated. Such branches go from immediately before any nodes after the earliest point that termination can occur in the process, and go to the close node of the process.

Each terminate statement and each process that can be prematurely terminated by that statement (i.e. each process specified in the statement along with each dependent son, grandson etc. of such processes) is considered in turn. A breadth-first search is performed from the entry node of the process that can be terminated to determine the earliest point on each path in that process at which the termination can occur. The earliest point is the first node that is encountered on the path for which the terminate statement does not belong to the AFTER set. Subsequently, branches are inserted before each node after (and including) that earliest point, going to the close node.

The insertion of these branches may affect the generated execution sequence sets, so these must be regenerated after all terminations have been considered. However, the regeneration of the execution sequence sets may, in turn, affect the earliest points at which termination can occur on some paths in some processes, so this must be rechecked. These two steps alternate until there are no more conditional branches that need to be inserted.

6.5

PROPERTIES OF THE EXECUTION SEQUENCE SETS

The algorithms for execution sequence set generation, spurious IPPE elimination, and taking account of potential premature terminations, imply certain properties that the execution sequence sets will have. This in turn implies certain things about the IPPEs that will be removed, and also about the extra conditional branches that will be inserted.

The simplifications we have applied to the problem (e.g. ignoring RTE-clock time, branch and loop conditions etc.) have all acted to increase the number of possible paths through each process, and also the number of possible execution sequences where processes are running in parallel. Were it somehow possible to take all factors into consideration and do a strictly accurate analysis, the number of execution sequences would be smaller and the execution sequence sets would, in many cases, contain different nodes to those that are actually generated by the algorithms in section 6.2. These sets would instead be the genuine execution sequence sets according to the rules in section 6.1.

In a graph containing no spurious IPPEs, the generated NEVER, BEFORE, AFTER, ALWAYS_BEFORE, and ALWAYS_AFTER sets will be subsets of the corresponding genuine execution sequence sets. The NEVER and ALWAYS sets are generated simply from the structure of the graph, assuming all combinations of paths are possible. Taking all factors into consideration reduces the number of such paths, and hence can only increase the size of those sets. The BEFORE sets are generated using the assumption that, at a particular wait node, for the wait to be satisfied requires only one term in each conjunct to be true, and hence for only one statement to have executed setting a term in the conjunct to true. The BEFORE set of the wait node contains only those nodes that can be guaranteed to execute before any of the nodes whose execution would set a term to true; i.e. before any of the nodes which are tails of IPPEs in a conjunct group at the wait node.

Taking all factors into consideration would reduce the number of possible execution sequences, thereby implying more forced orderings and larger BEFORE and AFTER sets. The ALWAYS_BEFORE and ALWAYS_AFTER sets are generated by taking the intersections of the ALWAYS sets with the BEFORE or AFTER sets; since these are subsets of the corresponding genuine execution sequence sets, the resulting ALWAYS_BEFORE and ALWAYS_AFTER sets must also be subsets of the corresponding genuine execution sequence sets.

The generated POSSIBLY_BEFORE and POSSIBLY_AFTER sets, on the other hand, are supersets of the corresponding genuine execution sequence sets. This can be seen more easily by a consideration of the generation algorithms themselves than by any other way. The POSSIBLY_BEFORE set at a wait node is generated by taking the union

of the POSSIBLY_BEFORE sets of all nodes whose execution can satisfy the wait expression - i.e. all nodes which are tails of IPPEs going to the wait node. The genuine POSSIBLY_BEFORE sets at the wait node would consist of a subset of the union of the POSSIBLY_BEFORE sets of only those nodes that may have to execute before the wait is completed. Clearly, even those nodes will be a subset of the ones that can satisfy the wait expression.

During IPPE elimination, the graph contains spurious IPPEs; i.e. IPPEs representing impossible execution sequences. At a wait node, the effect of a spurious IPPE directed towards that node is such as to further reduce the generated BEFORE set of the node, by adding one more set to be intersected with. Thus, spurious IPPEs reduce the size of the generated BEFORE sets (and hence the generated AFTER sets), although there is no effect on the generated ALWAYS and NEVER sets. Spurious IPPEs therefore reduce the size of the generated ALWAYS_BEFORE and ALWAYS_AFTER sets. Note that the generated BEFORE, AFTER, ALWAYS_BEFORE, and ALWAYS_AFTER sets are still subsets of the genuine execution sequence sets.

The effect of a spurious IPPE directed towards a wait node is such as to increase the size of that node's generated POSSIBLY_BEFORE set, by adding one more set to be unioned with. Hence the generated POSSIBLY_BEFORE and POSSIBLY_AFTER sets are still supersets of the corresponding genuine execution sequence sets, even with the presence of spurious IPPEs in the graph.

The algorithm for removing spurious IPPEs uses the generated execution sequence sets in such a way as to guarantee that only spurious IPPEs can be removed. There is, however, no guarantee that

all spurious IPPEs will be removed in this manner.

As far as the modifications to the graph that are necessary to account for potential premature terminations are concerned, the point that is calculated to be the earliest, on a path in a process at which termination can occur, is at least as early as the genuine earliest such point. This can be seen by the fact that the generated AFTER sets are subsets of the genuine AFTER sets, and since the node before the earliest termination point has the terminate statement in its AFTER set, the termination cannot possibly occur until after this point. It is assumed that termination can occur at any time after this calculated earliest point has been reached, which will clearly include all the points at which termination can actually occur.

The effect of introducing extra conditional branches into the IPPG is such as to reduce the size of the BEFORE, AFTER, and ALWAYS sets, by introducing extra execution paths and sequences. During the modification/execution-sequence-set-regeneration cycle, therefore, the earliest points at which termination can occur can only become earlier in a particular process, and consideration of the number of nodes in the processes being terminated will show that the number of times around this cycle will be finite and bounded.

The method of treating potentially premature terminations can be seen to preserve the property that the execution sequences apparently possible from a consideration of the graph are a superset of the genuinely possible execution sequences. This still implies, therefore, that the generated NEVER, ALWAYS, BEFORE, AFTER, ALWAYS_BEFORE, and ALWAYS_AFTER sets will be subsets of the corresponding genuine execution sequence sets, while the generated

POSSIBLY_BEFORE and POSSIBLY_AFTER sets will be supersets of the corresponding genuine execution sequence sets.

CHAPTER VII

SYNCHRONISATION ANOMALY DETECTION

The final execution sequence sets contain a large amount of information concerning the possible and forced orders of execution of the individual statements in the program under analysis. As such, they prove useful in the detection of data-flow anomalies concerning shared variables; see [5]. In this thesis, however, we are concerned with the detection of synchronisation anomalies. These also prove to be readily detectable from the execution sequence sets.

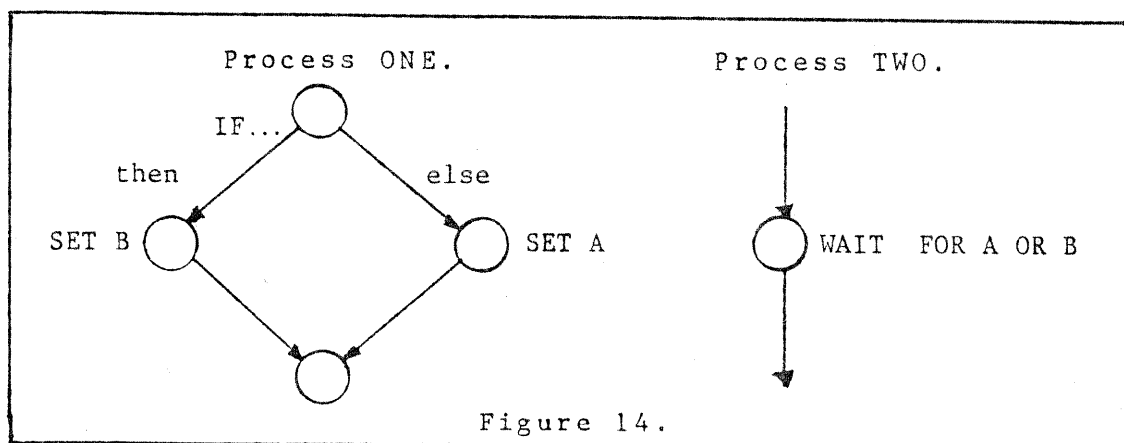
7.1

INFINITE WAITS

A process will wait indefinitely at a wait statement if the wait expression is false when the wait is reached, and one of two situations occurs. Either no combination of statements such as to set the expression to true remain to be executed in other processes, or all such combinations will be prevented from executing because the process under consideration is stalled. The latter case represents what is usually called a deadlock.

The detection method involves considering each wait statement in turn for the possibility of such a situation. The wait expression will already have been translated into conjunctive normal form during the building of the IPPG. For a potentially infinite wait, there must be at least one conjunct that can be false after all the statements have executed that occur before or concurrently with the

wait. (This is a necessary condition, but not a sufficient one - hence the situations for which this is true will include a superset of the actual potentially infinite waits in the program). This in turn requires that all the terms in that conjunct can be false. The algorithm takes a simplification of this by assuming that the conjunct must be true only if at least one particular term in the conjunct must be true; i.e. it does not take account of situations in which at least one term must be true, but there is no guarantee as to which one it will be. (see figure 14).



The check proceeds term by term through each conjunct until either a term is reached which must become true in finite time despite the wait - in which case the conjunct cannot remain indefinitely false and the next conjunct is checked - or until the end of the conjunct is reached without finding such a term - in which case it is assumed that the conjunct does not necessarily have to become true, and hence that there is an anomaly.

The worst case is assumed while checking a term. It is assumed that, for all nodes setting the term to true that can be prevented from executing until after the wait has completed (i.e.

those belonging to the POSSIBLY_AFTER set at the wait node), an execution sequence exists in which all such nodes will be so prevented. Further, of the remaining nodes setting the term to true (i.e. those that must execute before or in parallel with the wait), an execution path exists in which only those belonging to the ALWAYS set of the wait node will execute. Finally, it is assumed that all nodes setting the term to false not belonging to the AFTER or NEVER sets of the wait node can execute before the wait is reached. Given these assumptions, the term can be indefinitely false if either:

- no nodes remain setting the term to true that can execute before the wait has completed, or:

- each such node can be followed by a node setting the term to false, that is not itself always followed by a node setting the term to true.

Thus the wait at wait node W is potentially infinite if there exists a conjunct C in the wait expression such that:

For each term T in C and

For each node N_t setting T to true

N_t belonging to $\{ALWAYS(W) - POSSIBLY_AFTER(W)\}$

There exists a node N_f setting T to false

N_f belonging to $\{\text{all nodes} - \{NEVER(N_t) \text{ union } BEFORE(N_t)\}\}$

intersection $\{\text{all nodes} - \{NEVER(W) \text{ union } AFTER(W)\}\}$

And such that no node N_t exists setting T to true

N_t belonging to $\{ALWAYS_AFTER(N_f)\}$ intersection

$\{\text{all nodes} - \{NEVER(W) \text{ union } POSSIBLY_AFTER(W)\}\}$

The algorithm itself is a direct implementation of the above expression

By a consideration of the results in 6.5, it can be seen that $\{ALWAYS(W) - POSSIBLY_AFTER(W)\}$ is a subset of those nodes setting a term to true that must actually execute before, or concurrently with, the wait. Also, $\{\text{all nodes} - \{NEVER(N_t) \text{ union } BEFORE(N_t)\}\}$ intersection $\{\text{all nodes} - \{NEVER(W) \text{ union } AFTER(W)\}\}$ is a superset of those nodes setting a term to false that can execute after N_t and before the wait is completed. Finally, $\{ALWAYS_AFTER(N_f)\}$ intersection $\{\text{all nodes} - \{NEVER(W) \text{ union } POSSIBLY_AFTER(W)\}\}$ is a subset of those nodes setting the term to true that would actually execute after N_f and before the wait is completed.

Given these results, it can readily be seen that the algorithm identifies a superset of those terms that can be false for the duration of the wait. Hence the algorithm identifies a superset of the actual potentially infinite waits in the program under analysis.

7.2

RESCHEDULING A LIVE PROCESS

The method for detecting potential situations where a process may be rescheduled while still on the process queue from some previous schedule involves considering the execution sequence sets at each close and schedule node.

For a process P, if a schedule or close for one instantiation of P is in a set other than the NEVER, BEFORE, or AFTER sets at the schedule or close node for a different instantiation of P, there is an anomaly. Clearly, in such situations, the schedule/close of one instantiation of P can occur in parallel with the schedule/close of the other instantiation, and hence possibly in the wrong order. Since the BEFORE, AFTER, and NEVER sets are subsets of the

corresponding genuine execution sequence sets, the detected anomalies of this type are a superset of the actual anomalies present in the program.

One other possibility exists for an anomaly of this type, where there is no possible parallelism of the schedule and/or close nodes but the processes may still overlap. This is allowed for by checking, at each schedule node, that if a schedule for a different instantiation of the same process belongs to the BEFORE set, then the close node for that instantiation must also belong to the BEFORE set. If not, there is an anomaly. Due to the symmetry in the BEFORE and AFTER sets, the possibility that a different instantiation of the process will start after the schedule but before the close of this instantiation, is checked for when considering the schedule node for that instantiation itself. All such anomalies of this type are therefore found.

7.3 TERMINATING, CANCELLING, OR UPDATING THE PRIORITY OF

A DEAD PROCESS

The possibility of these anomalies is discovered by considering the execution sequence sets at each terminate, cancel, and update priority node (collectively referred to as T nodes). To guarantee that a process referred to at a T node is on the process queue, there must be a schedule for that process belonging to the ALWAYS_BEFORE set of the T node, such that the corresponding close node of the process is in the AFTER set of the T node. If this is not the case, there is an anomaly. Since the ALWAYS_BEFORE and AFTER sets are subsets of the corresponding genuine execution sequence sets, we find

a superset of the anomalies of this type.

7.4

TERMINATING AN INDEPENDENT PROCESS

This possibility is checked for at the same time as the anomalies in 7.3. If the T node is a terminate node, then the instantiation of the terminated process that is on the process queue at the time of termination must be a dependent of the terminating process. If not, there is an anomaly.

7.5

PREMATURE TERMINATION

Checking for this anomaly is carried out at the same time as for those anomalies in 7.3 and 7.4. If, at a terminate node, the close node of the process being terminated is not in the BEFORE set, the process can be prematurely terminated. Since the BEFORE sets are subsets of the genuine BEFORE sets, we find a superset of the anomalies of this type.

The algorithms for detecting synchronisation anomalies appear in appendix B.4.

CHAPTER VIII

HOW GOOD IS THE SYSTEM?

Throughout this paper, there have been informal proofs that the proposed anomaly detection system finds a superset of the actual anomalies in the program under analysis. (The one exception is the problem with loops, which we hope to be able to correct).

Currently, it is up to the user to identify which of the reported anomalies actually represent possible errors in the program. Clearly, the higher the proportion of reported anomalies that represent errors and possible errors, the 'better' the system. If the proportion is low, the time taken to check out all of the anomalies will be high compared to the benefits to be gained, and the system may end up costing more time than it saves.

So far this report has not addressed the issue of just how high the percentage of possible errors to reported anomalies might be. This is because the only practical way to get this information would be to produce a working system, and to gather information from it. However, a fairly good way is to consider just where the system produces wrong results.

8.1 SAFE SITUATIONS FALSELY IDENTIFIED AS ANOMALOUS

Perhaps the easiest way to identify where such situations can arise is to discuss each simplification we made, and also each step in the algorithms, in turn.

Our first simplification was to ignore process priorities and RTE-clock time. We feel, however, that for a developer to rely on processes of high priority running ahead of processes with lower priority, or to rely on sections of code completing (or not completing) in specific RTE-clock time intervals, is both poor programming practice and potentially dangerous. Examples of where such a reliance might cause errors are:

- i) In a multiprocessor environment, a process with a high priority may be suspended on its own processor while a process with a lower priority is executing on a different processor.
- ii) A process normally taking a small amount of time to execute may be suspended for an external interrupt or for a process with a higher priority.

We therefore feel that, even were it possible to accurately take account of RTE-clock time, process priorities, and the resulting reduction in the number of possible execution sequences, we would still like to warn the user of situations relying on them to avoid errors, in the event, say, that the program is transferred to a different environment.

Our next simplification was to ignore the conditional expressions governing branches and loops, and hence to assume that all paths generated by the flowgraph are executable. This is a real problem which plagues other static anomaly detection systems relying on data-flow analysis. In a study of the DAVE system it was discovered that about fifteen percent of the anomalies detected were on unexecutable paths [8]. Algorithms are being developed for detecting some classes of unexecutable paths [8], [9], and may prove

useful additions to this system in the future.

Our next simplification was to ignore all non synchronisation statements and variables. However, since these only affect the synchronisation anomalies by determining which paths can be taken through each process, and which resulting execution sequences are possible, they do not cause additional anomalies other than those already mentioned.

The following encodings do not appear to introduce additional anomalies, although we have not yet attempted to prove this:

- the encoding of a close statement as a wait for dependents followed by a simple close.
- the encoding of a process scheduled such that it immediately enters the stalled state as having the process execute a wait as its first instruction.
- the encoding of a process scheduled to execute cyclically as a loop within that process.
- the encoding of a return as a branch to the close statement.
- the encoding of a signal on an unlatched event variable as a set followed by a reset.

The method of dealing with signals on latched event variables can cause the addition of spurious anomalies. These occur because of the possibility of synchronisation statements in other processes executing between the set and the reset in our encoding, which cannot occur in practice, and also because of the possibility of taking the wrong branch in the encoding since we assume that either path can be taken. However, we have discovered that a signal statement cannot cause a wait that is potentially infinite without the presence of the

signal to be guaranteed finite as a result of the signal, given the assumptions we have already made. This is because the signal must either be executed after the wait has completed, in which case it cannot affect the wait, or it can potentially be executed before the wait has been reached, and hence would also not affect the wait. As far as the other anomalies are concerned, the signal statements do not affect the detection of such anomalies except in the determination of the execution sequence sets. Accordingly all anomalies introduced by the encodings of signal statements can be avoided by removing all nodes resulting from the encodings from both the execution sequence sets and the IPPG immediately following the generation of the final execution sequence sets. During generation of the execution sequence sets, the method of dealing with signal statements does accurately reflect their behaviour, and does not cause any additional deviations from the genuine execution sequence sets.

The method of dealing with potential premature terminations, particularly of processes containing synchronisation statements affecting non dependent processes, can introduce a potentially large number of spurious anomalies. However, we feel that such situations are very dangerous anyway, and would strongly caution developers against including such situations in their programs. As far as the analysis is concerned, we can only say that we anticipate such situations will be extremely rare in the type of environment we intend our system for use in.

Additional anomalies can appear because of the approximations necessary to have reasonable time bounds on the algorithms to remove spurious IPPEs, to generate the execution sequence sets, and to detect potentially infinite waits.

In all three of these stages, the additional anomalies appear primarily in two different situations. Firstly, synchronisation errors and anomalies can cause a large amount of error cascading, producing many spurious anomalies. This is an aspect we have not considered in any detail, but which should be examined in the future for methods to reduce the phenomenon.

The second situation is where sets and resets on the same event variable can potentially occur in parallel. In such situations the complexity of the problem is high, and our algorithms produce results that deviate further from the correct results.

However, in such situations even a wrong result by our system (i.e. indicating that there is an anomaly when the program is perfectly safe) can be useful in that it uncovers an area of the program where the synchronisation is complex, and careful hand checking is necessary to ensure the absence of potential errors. Furthermore, it indicates a situation where possibly an attempt should be made to simplify the synchronisation, to facilitate greater ease of maintainability of the program.

An additional consideration at this point is the type of programming environment the tool is intended for use in. The system has been developed specifically for NASA, to be integrated into the MUST system [11]. The aim of the MUST system is to provide an environment that allows for the rapid development of reliable,

maintainable code. We anticipate that in such an environment the synchronisation in the programs will be kept as simple as possible, and our system will be used more to detect genuine errors and to validate programs than to aid in debugging programs containing a large complex interweaving of synchronisation statements. Clearly it is useful in the former case and possibly not so useful in the latter.

The one additional area where anomalies can appear is from the system's failure to satisfactorily take account of sets and/or resets occurring on mutually exclusive paths at a branch (see figure 14). The next chapter contains some ideas as to how the system may be modified to reduce such false anomalies, but for the time being we can only reiterate that we expect such situations will be infrequent in the type of programming environment that our system is aimed at.

CHAPTER IX

FUTURE WORK

There are three main areas where we anticipate work could usefully be done in the future. The first such area is the incorporation of the improvements already mentioned in this thesis, along with improving time and space bounds, and enforcing less restrictions on the programs that can be handled. The second area is the expansion of the system to detect more anomalies, and also to provide other useful information. The third area is the production of similar systems for other concurrent languages.

9.1

IMPROVEMENTS TO THE SYSTEM

The most important improvement appears to be the correction of the current problem concerning synchronisation statements in loops within processes. One way this might be achieved would involve treating a loop as a single composite node in the early stages of the analysis, and generating execution sequence sets that imply forced or possible execution orderings involving complete loops (e.g. that a particular loop will have completed execution before a particular node in another process; or that two loops can execute concurrently etc.). Clearly a rigorously accurate analysis of this type is not possible, but it may be possible to make approximations that allow for the detection of a small superset of the actual anomalies in polynomial time.

Another approach might be to proceed exactly as has been specified, but to insert an additional step that would search for the remaining anomalies. Again, this additional step would find a superset of the actual anomalies. If it could be determined how such additional anomalies can occur, it should be possible to produce a polynomial time-bounded algorithm to do this.

Assuming that a suitable method can be found to analyse loops containing synchronisation statements, and hence to remove the current restrictions concerning loops, there would remain only two restrictions on the programs that can be successfully analysed. These are the restrictions concerning event variable arrays and name variables.

Methods do exist for handling arrays in static data-flow analysis systems, although they are not entirely satisfactory. One such method involves treating the entire array as a single composite variable whose value in general cannot be determined. Such a method could readily be incorporated into our system by assuming that wherever there is a reference in a wait statement to an undeterminable element of an event variable array, or to an element of the array whose value cannot be determined, the value of the variable is such that the term in the wait expression is false. This would still guarantee to find all potential errors, although there would potentially be a large number of false anomaly reports generated.

A similar assumption could also be made to allow for the incorporation of a method to handle name variables of type event variable and event variable array. Unfortunately, we cannot

currently envisage a suitable system to handle name variables of types program, function, procedure, and task.

There are several ways in which the current system could be modified to reduce the number of anomalies reported that are not potential errors. One such method might be the inclusion of algorithms for the detection of unexecutable paths. (see chapter 8).

Another such modification we envisage would allow us to take account of sets and/or resets that occur on mutually exclusive paths at a branch in a process. It should prove possible to collapse such paths into a single node such that the 'statement' represented by that node would be the setting to true of an event expression. To return to figure 14 (page 67), the single node to replace the section of process ONE would be, in effect, a 'SET A OR B'. Following execution of this node, the expression 'A OR B' must be true, and hence the wait in process TWO can be determined to be finite. Taking this a step further, it should prove possible to collapse whole sections of processes, containing sets and/or resets, into single nodes, such that the 'statement's represented by those nodes would be the setting to true of an event expression containing both ANDs and ORs. This would also save time and space in addition to providing better results.

Another such modification would be to improve the handling of processes scheduled with a cancellation criterion. We currently assume that any process that is scheduled with a cancellation criterion can be cancelled before it has executed even once. However, it is possible that the process may at least have to start its execution before it can be cancelled, and hence that it must

execute at least once. It should be possible to take account of this by determining the earliest point at which the cancellation can occur in the execution of that process. If that point proves to be before the process has started execution, then our assumption is valid. Otherwise we would know that the process must execute at least once. It should also prove possible to take account of the fact that the process remains on the process queue at least until after the cancellation criterion is satisfied, by inserting a wait for the cancellation criterion immediately before the close node in that process.

At no stage during the development of this system has much thought been given to improving its time and space bounds. Several methods immediately come to mind as to how this may be achieved.

Firstly, during IPPE elimination, and the insertion of additional branches to allow for premature terminations, the execution sequence sets are regenerated many times, although for the majority of nodes they do not change at each such regeneration. If it could be determined which sets at which nodes can have changed at each regeneration, these sets alone would have to be regenerated.

Secondly, it is possible to have redundant IPPEs in the graph; i.e. IPPEs which are not spurious, but whose presence does not carry any additional information. If these IPPEs can be removed, execution sequence set generation would be speeded up.

Thirdly, it is possible that the IPPG will contain processes that do not contain any synchronisation statements other than the close statement, and cannot be terminated, cancelled, or have their priorities updated. Such processes cannot cause any anomalies, and

do not affect the execution sequence sets in other processes, so they can be removed from the graph. Furthermore, their initial pair of IPPEs are removed, along with their schedules, and any terms in wait expressions involving their process events.

9.2

EXTENSIONS TO THE SYSTEM

There are several additional features that the system might include. Some such features might be:

Distinguishing between possible errors and guaranteed errors. This requires the generation of other execution sequence sets as discussed in our earlier report [5].

Allowing the user of the system to pass in additional information that can be used by the system to reduce the number of apparent execution sequences, and the number of safe situations which are falsely reported as anomalous. Such information might include known input value ranges (to aid in the determination of unexecutable paths), known inter-process orderings not evident from the code itself etc.

Allowing for an anomaly definition capability. For instance, it may be desirable that, in a given program, say, two particular processes should never run concurrently. Such information is readily available from the execution sequence sets. This capability could either be implemented with assertions placed in the code, or by using the system interactively.

Continuing this still further, there is much potentially useful information contained in the IPPG and the execution sequence sets. A useful additional feature would be to allow the user, on

request, to obtain some or all of this information.

9.3 SIMILAR SYSTEMS FOR OTHER CONCURRENT LANGUAGES

The techniques presented in this thesis have been applied specifically to the HAL/S programming language. HAL/S has proved to be a fortunate choice, since it contains a very wide range of synchronisation constructs. With the exception of name variables, all such constructs can be dealt with by our system.

Some other concurrent languages contain only a subset of the concurrent features available in HAL/S. Our system should prove to be immediately applicable to such languages, by performing modifications to the front end.

CHAPTER X

CONCLUSION

We have presented the design of a system for the static analysis of HAL/S programs to detect various synchronisation anomalies. We intend such a system to be used alongside data-flow analysis techniques of the types found in DAVE [1], for the detection of data usage anomalies.

The constraints we initially imposed on such a system have necessarily implied that the results it produces can only be approximations to the actual anomalies present in the program. However, at each stage we have given much thought to reducing the number of spurious anomalies that will be introduced while at the same time retaining all original anomalies in the program.

The inter process precedence graph representation seems ideally suited to its needs, as it captures all the relevant information in the program in a simple and accessible way. The execution sequence sets, too, are a practical and simple way to store the sequencing information which must necessarily be obtained from the program. Once this information is available, the algorithms themselves are reasonably straightforward.

We have met our original aims, although we realise that the work presented here is preliminary, and much additional work still remains to be done. We have identified a number of areas where such work could usefully be performed, and no doubt more such areas will appear.

It remains to be seen how accurate and useful the final system proves to be.

BIBLIOGRAPHY

- [1] - Fosdick, L.D., and Osterweil, L.J., "Data Flow Analysis in Software Reliability", Computing Surveys, vol. 8, no. 3, pp 305-330 (September 1976).
- [2] - "The HAL/S Language Specification", Intermetrics, Inc., Cambridge, Massachusetts (June 1976).
- [3] - Taylor, R.N., and Osterweil, L.J., "Anomaly Detection in Concurrent Software by Data Flow Analysis", University of Colorado Technical Report CU-CS-152-79 (April 1979).
- [4] - Reif, J.H., "Analysis of Communicating Processes", TR30, University of Rochester, Dept. of Computer Science, New York, (May 1978).
- [5] - Bristow, G., Drey, C., Edwards, B., and Riddle, W., "Design of a System for Anomaly Detection in HAL/S", University of Colorado Technical Report CU-CS-151-79 (March 1979).
- [6] - Saxena, A., "Static Detection of Deadlocks", University of Colorado Technical Report CU-CS-122-77 (November 1977).
- [7] - Peterson, J.L., "Petri Nets", Computing Surveys, vol. 9, no. 3, pp 223-252 (September 1977).
- [8] - Bollacker, L.A., "Detecting Unexecutable Paths Through Program Flowgraphs", (Master's Thesis), University of Colorado, Dept. of Computer Science (August 1979).
- [9] - Osterweil, L.J., "The Detection of Unexecutable Program Paths Through Static Data Flow Analysis", University of Colorado Technical Report CU-CS-110-77 (May 1977).

APPENDIX A

TO PROVE:

That to determine whether an arbitrary HAL/S program contains a situation where a process may be rescheduled while still active is at least as hard as an NP-complete problem.

PROOF:

By a reduction of satisfiability, which is known to be an NP-complete problem. Satisfiability is as follows:

Given a set S of boolean variables, $S_1 \dots S_n$, and an expression E over the variables in S , does a combination of true or false values exist for the variables in S such as would have the expression E true?

REDUCTION:

- i) produce a set of event variables $EV_1 \dots EV_n$, corresponding to the boolean variables $S_1 \dots S_n$ in satisfiability
- ii) produce a program MAIN and a task T as follows:

| | | | |
|-------|-----------------------|----|--------|
| MAIN: | PROGRAM; | T: | TASK |
| | IF ... | | . |
| | THEN SET EV_1 ; | | . |
| | ELSE RESET EV_1 ; | | . |
| | IF ... | | . |
| | THEN SET EV_2 ; | | . |
| | ELSE RESET EV_2 ; | | . |
| | . | | . |
| | . | | . |
| | IF ... | | . |
| | THEN SET EV_n ; | | . |
| | ELSE RESET EV_n ; | | . |
| | SCHEDULE T; | | . |
| | WAIT FOR EE OR NOT T; | | . |
| | SCHEDULE T; | | . |
| | CLOSE: | | CLOSE: |

where EE is the expression formed by substituting EV for S in E .

At each branch in MAIN either path can be taken, and hence each event variable can be either true or false. Since we assume that all combinations of paths are possible, clearly all combinations of values for the event variables are possible. This corresponds to being able to make any choices for the values of the variables in S in the satisfiability problem. The only way that the WAIT can be completed before T has completed its first execution is if the expression EE is true. EE can only be true, however, if the expression E is satisfiable. The only way that T can be rescheduled while it is still active, therefore, is if the expression E is satisfiable. Determining whether E is satisfiable is an NP-complete problem, and hence determining whether this HAL/S program contains an anomaly is an NP-complete problem.

APPENDIX B

PSEUDO-CODE FOR THE SYSTEM

The pseudo-code in this appendix uses the syntax defined in Caine and Gordon, "PDL - A tool for software design", PROC 1975 National Computer Conference, June 1975 pp. 271-276.

SEGMENT build IPPG

| | |
|--|---------|
| Focus | page 91 |
| Insert MAIN | 95 |
| inject | 97 |
| remove spurious IPPEs | 109 |
| calculate execution sequence sets | 98 |
| find potentially infinite waits | 110 |
| find instances of processes being rescheduled while still active | 112 |
| find other anomalies | 113 |

END

SEGMENT Focus

```
*****
*
*   This segment inserts subprograms in-line, removes non
*   synchronisation statements, and transforms synchronisation
*   statements as in 5.1.
*
*****
```

DO for each subprogram S in leafs-up order, and subsequently
each program and task

```
*****
*
*   Insert subroutines in-line where necessary
*
*****
```

DO for each node n in S

IF n is a call to a subprogram P

IF flowgraph for P is empty

DO for each FOCE (n_i, n)

DO for each FOCE (n, n_j)

create FOCE (n_i, n_j)

ENDDO

delete (n_i, n)

ENDDO

DO for each FOCE (n, n_j)

delete (n, n_j)

ENDDO

delete n

ELSE create new copy C of flowgraph for P

DO for each FOCE (n_i, n)

create FOCE (n_i, C_e)

```
*****
*
*      Ce is the entry node of C      *
*
*****
```

delete (n_i, n)

ENDDO

DO for each FOCE (n, n_j)

create FOCE (C_e, n_j)

```
*****
*
*      Ce is the exit node of C      *
*
*****
```

delete (n, n_j)

ENDDO

delete n

ENDIF

ENDIF

ENDDO

```

*****
*
*          Remove all non-synchronisation statements
*
*****

```

```

DO   for each node n in S

      IF   n is a synchronisation statement, or the entry
            node S

            DO   for the first synchronisation node or
                  exit node,  $n_f$ , that is encountered on
                  each path in a breadth-first search
                  from n

                  IF   there does not exist an FOCE  $(n, n_f)$ 
                        create an FOCE  $(n, n_f)$ 

                  ENDIF

            ENDDO

      ENDIF

ENDDO

DO   for each node n in S that is not a synchronisation
      node or the entry or exit node

      delete all FOCEs into n
      delete all FOCEs leaving n
      delete n

ENDDO

```

IF the flowgraph for S contains only S_e and S_x

set the flowgraph for S to null

```
*****
*
* otherwise, remove all loops and transform
* synchronisation statements as in 5.1.
*
*****
```

ELSE build a breadth-first search tree of the subprogram

DO for each back edge (n_i, n_j) in reverse order of
the length (no. of levels) of the back edge

create new copy of the portion of the flowgraph
consisting of nodes n_i, n_j , and all nodes on
forward paths from n_i to n_j

DO for each edge (n_k, n_l) going from a node in
the section that had been duplicated to a
node outside that section

add FOCE from new n_k to n_l

ENDDO

add FOCE from n_j to new n_i
delete foce from n_i to n_j

ENDDO

transform synchronisation statements (except schedule
statements) as specified in 5.1.

ENDIF

ENDDO

END

SEGMENT Insert (P : process) at (n : node) in (Q : process)

* This segment inserts a new copy of the flowgraph for P
* into the IPPG where a schedule appears at node n in
* process Q.
*

create new copy C of flowgraph for P

IF n has a repeat clause in the schedule

create new node m

IF C has a wait for dependents, w_d , inserted immediately
before the close statement

DO for each FOCE (1, w_d)
create FOCE (1, m)
delete FOCE (1, w_d)

ENDDO

create FOCE (m, w_d)

ELSE DO for each FOCE (1, C_x) where C_x is the close node
of C

create FOCE (1, m)
delete FOCE (1, C_x)

ENDDO

create FOCE (m, C_x)

ENDIF

create FOCE (m, C_e) where C_e is the entry node for C

expand C as in segment focus

ENDIF

add c to the IPPG

create IPPE (n, C_e)

IF the schedule specifies P to be dependent
 create IPPE (C_x, Q_w) where Q_w is the wait for dependents
 immediately before the close of Q
ELSE create IPPE (C_x, R_w) where R_w is the wait for dependents
 immediately before the close of the
 enclosing program

ENDIF

IF the schedule has a stalled clause until expression E is true
 create node k of type WAIT FOR E

DO for each FOCE ($C_e, 1$)
 create FOCE (k, 1)
 delete FOCE ($C_e, 1$)

ENDDO

create FOCE (C_e, k)

ENDIF

DO for each schedule node S in C, on process A
 INSERT (A) at (S) in (C)

ENDDO

END

SEGMENT Inject

```
*****  
*                                                                 *  
*           This segment inserts all IPPEs that could be needed *  
*                                                                 *  
*****
```

```
  DO   for every wait node w in the graph  
        translate the wait expression into conjunctive normal form  
        DO   for each term t in the wait expression  
              DO   for every node n whose execution would set t true  
                    create IPPE (n,w)  
              ENDDO  
        ENDDO  
  ENDDO  
END
```

SEGMENT calculate execution sequence sets

calculate ALWAYS sets
calculate NEVER sets

DO for all nodes in the flow graph

BEFORE(n) = \emptyset

ENDDO

DO for all nodes n in the graph in breadth-first search order

BEFORE(n) = CALCULATE_BEFORE(n,p)

ENDDO

calculate AFTER sets
calculate ALWAYS_BEFORE and ALWAYS_AFTER sets
calculate POSSIBLY_BEFORE and POSSIBLY_AFTER sets

END

SEGMENT calculate ALWAYS sets

calculate ALWAYS_BEFORE_WITHIN_PROCESS and BEFORE_WITHIN_PROCESS
sets
calculate ALWAYS_AFTER_WITHIN_PROCESS and AFTER_WITHIN_PROCESS
sets
calculate ALWAYS_SUBPROCESS and SUBPROCESS sets
calculate final ALWAYS sets

END

SEGMENT calculate NEVER sets

calculate NEVER_WITHIN_PROCESS sets
calculate final NEVER sets

END

```

SEGMENT   calculate ALWAYS_BEFORE_WITHIN_PROCESS and
              BEFORE_WITHIN_PROCESS sets

  DO       for all processes p in the flow graph

            DO   for all nodes n in p in breadth-first search order

              ABWP = ALWAYS_BEFORE_WITHIN_PROCESS(e) for any node e
                  such that there exists an FOCE (e,n)

              BWP  = BEFORE_WITHIN_PROCESS(e)

            DO   for all nodes f ≠ e such that there exists an
                  FOCE (f,n)

              ABWP = ABWP intersection ALWAYS_BEFORE_WITHIN_
                    PROCESS(f)
              BWP  = BWP intersection BEFORE_WITHIN_PROCESS(f)

            ENDDO

              ALWAYS_BEFORE_WITHIN_PROCESS(n) = ABWP
              BEFORE_WITHIN_PROCESS(n) = BWP

            ENDDO

  ENDDO

END

```

SEGMENT calculate ALWAYS_AFTER_WITHIN_PROCESS and
AFTER_WITHIN_PROCESS

DO for all processes p in the flowgraph

DO for all nodes n in p in bottom-up breadth-first
search order

AAWP = ALWAYS_AFTER_WITHIN_PROCESS(e) for any node e
such that there exists an FOCE (n,e)

AWP = AFTER_WITHIN_PROCESS(e)

DO for all nodes f \neq e such that there exists an
FOCE (n,f)

AAWP = AAWP intersection ALWAYS_AFTER_WITHIN_
PROCESS(f)

AWP + AWP union AFTER_WITHIN_PROCESS(f)

ENDDO

ALWAYS_AFTER_WITHIN_PROCESS(n) = AAWP
AFTER_WITHIN_PROCESS(n) = AWP

ENDDO

ENDDO

END

```
SEGMENT calculate ALWAYS_SUBPROCESS and SUBPROCESS sets

  DO for each process p in the flowgraph in leafs-up order

    ASS =  $\emptyset$ 

    DO for all processes q such that SCHEDULE (q) belongs to
      ALWAYS_AFTER_WITHIN_PROCESS( $p_e$ )

      ASS = ASS union ALWAYS_SUBPROCESS(q)
      ASS = ASS union q

    ENDDO

    ALWAYS_SUBPROCESS(p) = ASS

    SS =  $\emptyset$ 

    DO for all processes q such that SCHEDULE (q) belongs to
      AFTER_WITHIN_PROCESS( $p_e$ )

      SS = SS union SUBPROCESS(q)
      SS = SS union q

    ENDDO

    SUBPROCESS(p) = SS

  ENDDO

END
```

```

SEGMENT calculate final ALWAYS sets
  DO for each process p in the flowgraph in breadth-first order
    IF p is the main program
      A = pe
      A = A union ALWAYS_AFTER_WITHIN_PROCESS(pe)
      DO for all processes q belonging to ALWAYS_SUBPROCESS
        set of p
          A = A union (qe)
          A = A union ALWAYS_AFTER_WITHIN_PROCESS(qe)
      ENDDO
      ALWAYS(pe) = A
    ELSE ALWAYS(pe) = ALWAYS(SCHEDULE p node)
    ENDIF
  DO for all nodes n in p excluding pe
    A = n
    A = A union ALWAYS_AFTER_WITHIN_PROCESS(n)
    A = A union ALWAYS_BEFORE_WITHIN_PROCESS(n)
    A = A union ALWAYS(pe)
    DO for each process q such that SCHEDULE (q) belongs
      to A minus ALWAYS(pe)
      DO for each process r belonging to ALWAYS_
        SUBPROCESS(q)
          A = A union (re)
          A = A union ALWAYS_AFTER_WITHIN_PROCESS(re)
      ENDDO
      A = A union (qe)
      A = A union ALWAYS_AFTER_WITHIN_PROCESS(qe)
    ENDDO
    ALWAYS(n) = A
  ENDDO
ENDDO
END

```

```
SEGMENT calculate NEVER_WITHIN_PROCESS sets
  DO for all processes p in the flowgraph
    DO for all nodes n in p
      NWP = all nodes in p
      NWP = NWP minus BEFORE_WITHIN_PROCESS(n)
      NWP = NWP minus AFTER_WITHIN_PROCESS(n)
      NWP = NWP minus n
      NEVER_WITHIN_PROCESS(n) = NWP
    ENDDO
  ENDDO
END
```

```

SEGMENT calculate final NEVER sets
  DO for each process p in the flowgraph in breadth-first order
    IF p is not the main program
      NEVER( $p_e$ ) = NEVER( SCHEDULE (p) node)
    ENDIF
    DO for each node  $n \neq p_e$  in p
      N = NEVER( $p_e$ )
      N = N union NEVER_WITHIN_PROCESS(n)
      DO for each process q such that SCHEDULE (q) belongs
        NEVER_WITHIN_PROCESS(n)
        N = N union all nodes in q
        DO for each process r belonging to SUBPROCESS(q)
          N = N union all nodes in r
        ENDDO
      ENDDO
      NEVER(n) = N
    ENDDO
  ENDDO
END

```


SEGMENT CALCULATE_BEFORE(m,p)

```
*****
*
*           m is a node in process p           *
*
*
*****
```

$N = \{\text{all nodes}\} = \text{identity}$

IF there exists (n,p) on SAVE such that n belongs to NEVER(m)
union {ancestors of m }

 REDO = all entries on SAVE from (n,p) to the top

CALCULATE_BEFORE = N

RETURN

ENDIF

IF BEFORE(m) = \emptyset

CALCULATE_BEFORE = BEFORE(m)

RETURN

ENDIF

push (m,p) onto SAVE

BEFORE_SET = BEFORE_WITHIN_PROCESS(m)

E = the union of all nodes e in p such that there exists a flow
graph edge (e,n)

IF $E \neq \emptyset$

 TEMP = N

DO for all nodes e belonging to E

 TEMP = TEMP intersection {CALCULATE_BEFORE(e,p) union
 NEVER(e)}

ENDDO

I = the union of all nodes k in process q such that there
exists an IPPE (k,m)

IF $I \neq \emptyset$

 TEMPX = N

DO for all nodes k belonging to I

 TEMPX = TEMPX intersection {CALCULATE_BEFORE(k,q)
 union k union NEVER(k)}

ENDDO

ENDIF

ENDIF

BEFORE_SET = BEFORE_SET union TEMP union TEMPX

IF m belongs to REDO

BEFORE(m) = BEFORE_SET

ELSE remove m from REDO

ENDIF

pop SAVE

END

```
SEGMENT calculate AFTER sets  
  DO for all nodes n in the flowgraph  
    DO for all nodes m belonging to BEFORE(n)  
      AFTER(m) = AFTER(m) union n  
    ENDDO  
  ENDDO  
END
```



```
SEGMENT calculate ALWAYS_BEFORE and ALWAYS_AFTER sets  
  DO for all nodes n in the flowgraph  
    ALWAYS_BEFORE(n) = ALWAYS(n) intersection BEFORE(n)  
    ALWAYS_AFTER(n) = ALWAYS(n) intersection AFTER(n)  
  ENDDO  
END
```

```
SEGMENT calculate POSSIBLY_BEFORE and POSSIBLY_AFTER sets

  DO for all nodes n in the flowgraph

    POSSIBLY_BEFORE(n) = BEFORE(n)
    POSSIBLY_AFTER(n) =  $\emptyset$ 

  ENDDO

  DO for all nodes n in the flowgraph in breadth-first search
  order

    DO for all nodes e such that there exists an edge (e,n)

      POSSIBLY_BEFORE(n) = POSSIBLY_BEFORE(n) union e union
      POSSIBLY_BEFORE(e)

    ENDDO

  ENDDO

  DO for all nodes n in the flowgraph

    DO for all nodes m belonging to POSSIBLY_BEFORE(n)

      POSSIBLY_AFTER(m) = POSSIBLY_AFTER(m) union n

    ENDDO

  ENDDO

END
```

```
SEGMENT  remove spurious IPPEs

  DO  until no more spurious IPPEs can be removed

        calculate BEFORE sets

        calculate AFTER sets

        calculate ALWAYS_AFTER sets

        calculate NEVER sets

  DO  for each IPPE in the flowgraph

        IF  predecessor node belongs to AFTER(successor node)
        OR  predecessor node belongs to NEVER(successor node)
        OR  there exists a node negating the predecessor node
            belonging to ALWAYS_AFTER(predecessor node) union
            BEFORE(successor node)

            remove IPPE from the flowgraph

        ENDIF

  ENDDO

ENDDO

END
```

```

SEGMENT find potentially infinite waits
  DO for each wait node w in the graph
    POSSIBLY_INFINITY = FALSE
    DO for each conjunct c in w
      CONJUNCT_POSSIBLY_INFINITY = TRUE
      DO for each term t in c
        TERM_POSSIBLY_INFINITY = TRUE
        DO for each node  $n_t$  setting t to true belonging
          to ALWAYS(w) minus POSSIBLY_AFTER(w)
          NODE_POSSIBLY_NEGATED = FALSE
          DO for each node  $n_f$  setting t to false
            belonging to {all nodes minus BEFORE( $n_t$ )
              union NEVER( $n_t$ )} intersection {all nodes
              minus {NEVER(w) union AFTER(w)}}
            IF there does not exist a node  $n_t$ 
              setting t to true belonging to
              {ALWAYS_AFTER(w) intersection
              {all nodes minus {NEVER(w) union
              POSSIBLY_AFTER(w)}}}
              NODE_POSSIBLY_NEGATED = TRUE
          ENDIF
        ENDDO
      IF NODE_POSSIBLY_NEGATED = FALSE
        TERM_POSSIBLY_INFINITY = FALSE
      ENDIF
    ENDIF
  ENDIF

```

```
      IF   TERM_POSSIBLY_INFINITY = FALSE
            CONJUNCT_POSSIBLY_INFINITY = FALSE
      ENDIF
    ENDDO
    IF   CONJUNCT_POSSIBLY_INFINITY = TRUE
            POSSIBLY_INFINITY = TRUE
    ENDIF
  ENDDO
IF   POSSIBLY_INFINITY = TRUE
      write error message
ENDIF
ENDDO
END
```

```
SEGMENT find instances of processes being rescheduled while still
active

  DO for each SCHEDULE (p) in the flowgraph

    IF there exists a different SCHEDULE (p) or CLOSE (p)
      belonging to {all nodes minus {BEFORE union AFTER
      union NEVER sets for this SCHEDULE (p)}}

      write error message

    ENDIF

    IF there exists a different SCHEDULE (p) belonging to
      BEFORE (this SCHEDULE (p))

    AND the corresponding CLOSE (p) does not belong to
      BEFORE (this SCHEDULE (p))

      write error message

    ENDIF

  ENDDO

END
```



```
SEGMENT find other anomalies  
  
  DO for each T node in the flowgraph  
    DO for each process p that may be affected by the T node  
      IF SCHEDULE (p) does not belong to ALWAYS_BEFORE(T)  
      OR CLOSE (p) does not belong to AFTER(T)  
        write error message  
      ENDIF  
      IF the T node is a terminate node  
      IF T does not belong to AFTER (CLOSE (p))  
        write error message  
      ENDIF  
      IF p is not dependent on the process containing T  
        write error message  
      ENDIF  
    ENDIF  
  ENDDO  
  
ENDDO  
  
END
```

