Simulation Effectiveness[1] and Design Verification (Final Report)

Technical Report CRI-85-33
(DISC 84-2 - October 1984)

Alice C. Parker, Principal Investigator
Nohbyung Park and David W. Knapp

---

[1]This research was supported by International Business Machines Corporation Contract # S 956501 Q LX A B22.

Table of Contents

List of Figures

List of Tables

Abstract

This final report summarizes research performed from 1981-1984 on the topic of simulation effectiveness and design verification. The goal of this project was to aid in production of designs with fewer errors.

The first stage of the research was to survey designers about errors, and classify errors into types. This classification resulted in three types of errors, logical, timing and concurrent errors. The rest of the research focused on concurrent errors, since they were the most troublesome and difficult to detect.

A literature survey and study of alternative approaches to the problem formed the second stage of the research. While the literature did not reveal any techniques directly applicable to the problem, it became apparent that the state of the art in formal models of behavior was not adequate for this task. In addition, symbolic simulation did not take into account concurrent behavior. Thus, new techniques and theory would have to be developed.

The third phase of the research concentrated on a single approach to the problem - correctness by construction. A representation for design data was developed, and two experiments were carried out. The first experiment involved writing and testing a program which determines potential resource conflicts. The second experiment involved a program which retimes existing logic or synthesizes a clocking scheme for an incomplete design. This program finds a solution guaranteed to meet timing constraints with no errors in operation.

Two themes became prominent during this research: the necessity of control-flow and data-flow analyses, and the importance of proper representation of design information in order to perform the analyses. In addition, the types of errors which are difficult to detect cannot be detected easily using current simulation techniques. Therefore, other types of analyses and more sophisticated design capture techniques must be used to reduce the likelihood of design errors.

1 Introduction

The goal of the research described here has been to verify correctness of digital designs. We consider a design to be **correct** if we have proven that it is free from all design errors. Because total correctness is difficult (and sometimes impossible) to achieve, the scope of the investigation has involved partial correctness - a freedom from certain classes of design errors. While the results of this study have given us insight into the formal proof process, the study can also be used to answer the question, "how much simulation is enough to satisfactorily verify design correctness?" In specific, the study sought to investigate whether there were practical indicators that could measure the level of confidence that could be assigned to a given design and set of simulation test cases.

The study began with a survey of design errors and a classification of these into error types. Examples of these errors were given in a hardware-descriptive language, and specified formally using Behavior Expressions, illustrating how the errors manifested themselves in Behavior Expressions. Informal proofs were done to determine under what circumstances the errors could not be detected in the Behavior Expressions. Some errors were also simulated using the SLIDE simulator. The results of this part of the study are detailed in [Parker 83].

The second phase of the study continued a broad literature survey along with an investigation of potential error detection techniques. The literature survey is found in Appendix 1. Section 2 describes the phase-two research.

The third phase of the research involved the construction of an expert system which supports correctness by construction as well as a number of static and dynamic error checks. While the system is far from complete, parts of the system have been implemented. An overview of the system is given in Section 3. The design representation has been specified and programmed, and a user interface constructed. This representation is described in along with one static check, the collision detector, in Section 4. Documentation and program listings are found in Appendix 2. Clocking analysis and synthesis have also been researched. Optimal clocking synthesis algorithms have been developed, giving us an example of the use of the expert system to perform correctness by construction. This research is reported in [Park 84]. The program listing and documentation are found in Appendix 3.

A side effect of this research project was an exploration of the relationship between synthesis and verification in [Parker 84]. Conclusions to the entire study are found in Section 5.

2 Assessment of Simulation Effectiveness

This section presents some conclusions and opinions about the present state of knowledge concerning simulation effectiveness. It also proposes attacks on the problem which would lead to short-term solutions to the problem.

2.1 Literature Survey Conclusions

One of the general conclusions we came to after performing the literature survey is that there are basic differences between the concurrent behavior of hardware and software. The effect of this is that concurrent software research tends to focus on issues that are not relevant to hardware concurrency, and common issues become simpler in the software domain due to restrictions on simultaneity, timing and program structure. A comparison of software and hardware concurrency is shown in Table 1.

The literature survey was done over a number of categories. We address our assessment of this literature using this same categorization.

There have been no significant publications except those from IBM which describe methods for selection of test data inputs for simulation. The current approach seems to be to use hardware simulation engines to increase error coverage.

There has been a large amount of research concerning formal specification of hardware. McFarland's Behavior Expressions [McFarland 83] model the same situations that many of the other models address. Few of the models seems to provide insight into the description of correct concurrent behavior. (Milner's synchronous calculus of communicating systems [Milner 82] may give us some insight but the research is theoretical and it is not clear at this time how applicable his model is.) Moskowski [Moszkowski 82] proposes to model some timing and sequencing phenomena which we would like to describe but the research seems to be in the early stages and hence cannot be directly applied to our problem.

Hardware timing errors are the area of hardware errors most well understood. IBM is the leader in this research area [Hitchcock 82], but some different approaches have been proposed, such as symbolic simulation with timing predicates by Pitchumani [Pitchumani 82].

Hardware verification research has evolved in two directions: formal models which allow us to reason about the designs, and symbolic simulation. The mechanisms for verification differ between these two techniques, and their appeal differs. Symbolic simulation is more developed as a research field, but the abstraction process involved in formal models allows us the opportunity to discard, digest or encapsulate irrelevant information. The combination of abstraction with the techniques of symbolic simulation may well provide a future direction.

Table 1: CONCURRENT PROCESS DIFFERENCES : Software vs. Hardware

Software	Hardware
Flexible data structures (coroutines, heaps, stacks)	Tight constraints on data structures (depends on implementation)
Recursive execution; Re-entrant programming	No recursive execution; No re-entrant execution
Events in each process are synchronized to instruction execution cycle	Events can be completely asynchronous (e.g., Reset, Delay)
Explicitly specified boundaries between concurrent processes	Hard to partition a system into distinct concurrent processes
Correctness with respect to mapping of input values to output values (value mapping)	Correctness with respect to value mapping as well as input sequence of values/events to output sequence of values/events (sequence mapping) and timing
Structured process interaction; No asynchronous process initiation or termination	Unstructured process interaction; Processes can be started and killed asynchronously
Process interaction through variables(e.g. global variables semaphores, messages)	Process interaction through variables as well as controls over processes (e.g. reset, initiate, and terminate)

All of the hardware verification research reviewed is relevant and immediately useful in understanding the error detection process. However, there are limitations in practice which will be discussed in the next section.

Hardware design errors *per se* had not been investigated previously as a research area.

Hardware validation has seldom been the subject of research publications. One paper from IBM is interesting because it proposes hardware prototyping as a better technique for good error coverage than simulation [Tran 82].

Concurrent hardware research has focused on protocol modeling and distributed

processing. The level of detail and specification of timing used in the published research are inadequate for description of the kinds of errors which we would like to detect.

Software verification research has produced many interesting techniques but they cannot be applied directly to hardware verification. Rather, these techniques give us a foundation to build on for future research, because they are not powerful enough to handle the more complex problem of hardware verification. The main problem seems to be that hardware descriptions contain "go-to's" and that timing, sequencing, and concurrency are less restricted in hardware than in software. Finally, software verification addresses many types of errors which we are less interested in because they seem easier to detect by conventional methods.

Software test data selection is an active research field but either the research is not far enough along for us to use (Goodenough and Gerhart) [Goodenough 75] or the types of errors are the easy subset of the errors we are looking to detect.

Some software correctness research is useful to us. The Cause and Assert pair proposed by Feldman is useful for description of process synchronization [Feldman 79]. Flow expressions [Shaw 78] may be able to be combined with Behavior Expressions to produce Concurrent Behavior Expressions.

The software error research surveyed is useful to us because it gives us confidence that our hardware error classification is analogous. Three conclusions reached by software researchers are important. First, most errors occur because the program logic is not complex enough, meaning some code is omitted. (If this were also true of hardware, it would produce errors which could be difficult to detect.) Second, program analysis tends to detect errors earlier than execution does. (If this were true of hardware descriptions that would mean HDL analysis was better than simulation.) Third, many different methods are required to detect different types of errors, and there are many different reasons for those errors. Our suspicion is that this is even more true for hardware errors, because the range of types is broader than for software errors.

2.2 Problem Approaches

Due to the breadth of our problem statement and the lack of prior research in the field, much effort went into the exploration of alternative approaches to the problem and the selection of the current approach.

The attack on the problem has been three-pronged. The lines of reasoning are as follows:

1. either we can characterize error types adequately so that simulation test inputs can be selected on the basis of looking for certain types of errors;

2. *or* we can extract enough information from the designer so that we can form relatively complete assertions about the correct behavior to detect the errors by formal proof methods;

3. *or* we can extract from the description an abstract behavior and ask the designer if this is what he or she meant.

We have some conclusions about each of these approaches.

First, the characterization of errors led us to the conclusion that persistent errors are of many types, and we suspect that a large portion of the errors which are difficult to detect are due to oversimplification of the problem by the designer. They can be characterized loosely as "missing logic".

The remaining errors which we can characterize as mutations or other distinct types of errors have only been characterized qualitatively. Furthermore, according to the original error classification we produced, the wide range of error types and variety of places of occurrence force us to consider error coverage on a per-error-instance basis or by using estimates. Any simulation error coverage figures which could be produced would therefore be statistical rather than deterministic. In order to provide these error coverage measures, we would need a statistical model of error occurrences, which is probably best done in industry (i.e. within IBM). Fault testing research cannot be applied to error testing until error models analogous to stuck-at-one and stuck-at-zero can be used.

Second, the production of assertions for symbolic simulation seems to have the same correctness problems as production of the original hardware description. Furthermore, symbolic simulation does not address concurrent process errors at all. (A recent paper by V. Pitchumani indicates the difficulty of proofs when process communication and synchronization are unstructured [Pitchumani 84].) Thus, symbolic simulation *per se* does not appear to be immediately useful in addressing this problem. The equivalent of symbolic simulation can be done more easily and efficiently with the proper choice of design representation (e.g. our data structure).

The extraction of abstract behavior from the hardware descriptions to be fed back to the designer seems to have the most promise as an error detection technique for the long term. The derivation of both data flow (data precedence relationships) and control flow can be used directly to show the designer what he has specified in a different way, or can be further processed for more automatic detection of some errors.*The use of data flow analysis and/or specification of data flow by the designer seems to be a key part of any error detection strategy.*

Unfortunately, the extraction of abstract behavior in the form of Behavior Expressions will not address the problem of concurrent errors until we have a way of expressing

correct concurrent behavior, as discussed above. Furthermore, Behavior Expressions have some of the same problems as symbolic simulation because they quickly become complicated. Also, if a designer has oversimplified the problem, then he will probably view the oversimplified extracted behavior as correct. Finally, we cannot currently prove that two Behavior Expressions are NOT equivalent. The only way we can do this is either to specify canonical forms for all Behavior Expressions to be reduced to, or to prove non-equivalence by finding some set of simulation test inputs which lead to two different behaviors.

What we *can* do using the first approach is to show *how* to derive test case inputs or detect errors directly *given that a certain error is to be detected, and the circumstances of its occurrence are well characterized.* We do this in later sections of the report.

2.3 A Short-Term Solution

There is a short-term solution to the problem of selection of simulation test inputs for detection of errors. This solution avoids the error characterization problem. It assumes that the error type to be detected and the circumstances of occurrence have already been determined. It alleviates the problem of combinatoric explosion of test cases by determining *a priori* which are the cases of interest. The amount of preprocessing of the description is substantial, but more mechanical than the theorem-proving process which occurs during formal verification.

There is one constraint on this technique: it relies on being able to process the data flow and control flow of the HDL description[1]. Using this method, we can construct a control graph in such a way that it can explicitly show all the parallelism as well as the conditional branches in the control flow. Such control flow graphs have been widely described in the literature [Allen 76], [Dasgupta 76], [Andrews 80] and [Nagle 81].

For a procedural HDL like ISPS or SLIDE, extraction of the control flow and data flow is straightforward [McFarland 78]. For a non-procedural language like DDL, extraction of the control flow is more difficult, but not impossible.

Here is an example of the technique using the control graph constructed above. Suppose that one wanted to determine whether any registers were potential targets for value collisions, *i.e.* were being used to store more than one value at a time. Due to the combinatorics of looking for actual value collisions mechanically, we partition the problem by looking for possible problem locations. Cases of potential value collision include write-write and read-write across two parallel branches.

---

[1]The data flow contains the data precedence relationships - the dependency of values on other values, and the operations which produce them. The control flow specifies the order of events and the potential parallelism of events.

An example output trace could contain the following information: "register A is read on one side a parallel branch and written on the other side, but it is only read when C is 3 and only written when D is 4. C is the sum of P and Q, and Q is the difference of R and S. D is the sum of R and S. P, R and S are inputs to the description."

The user could then determine the simulation inputs required to get the correct conditional branches to be taken. Determination of the test inputs could also be done automatically, using an algorithm like the D algorithm to sensitize the proper conditional branches. This type of error detection is described further in Section 4.

A second type of error is the failure to restore state after exit from some exception handler. This could be detected by processing the description forwards, looking for variables written on only one side of a conditional branch. We already have a good algorithm to test whether a variable is written in every conditional path in a subgraph or not. The algorithm works for any kind of cyclic program graph. Working forwards, under guidance from the user, the program could then determine when or if a control variable (used later in a condition or predicate), would depend on the chosen variable. If so, then the program would work backwards from the point where the chosen variable is written to create the string of branch conditions and the history of branch variables in order to get the variable written and then used. This could then be used by the user to select test inputs for simulation.

Both these examples have things in common. First, the program is told which error to look for. Second, the error sites are identified as potential errors by detecting combinations of two events which may or may not be significant. Third, the user selects a particular site to examine. Fourth, the program works backwards to get a history of the conditions and conditional variables which could produce the detected combination of events. Finally, the user can examine the history in order to determine simulation inputs.

The above technique can be directly applied to the problem of simulation effectiveness. However, two other techniques can also be explored. They both involve data flow and control flow. The first technique involves feeding back both flows as derived from the description to the designer. The designer may then notice errors or anomalies not evident from the HDL itself. The second technique is to have the designer provide the flow information separately from the description itself. This would provide (in some sense) the specification of correct behavior independent of the description.

3 A Prototype System for Logical Error Detection

This section describes a prototype error detection system, the Designer's Apprentice, which aids designers in detection of logical design errors. This system is imbedded in the ADAM (Advanced Design AutoMation) system. It is intended to aid designers in two major ways: first, by reducing the probability of design errors; and second, by making

the design process more nearly automatic. This section will concentrate on the first aspect of the task.

The apprentice operates from designers' assumptions about the design (design practice) and rules about how the design must behave. These assumptions and rules may be specific to a particular architecture, or to a specific aspect of that architecture. For example, the assumption that instructions of a particular type always reset a certain flag when done is a rule specific to the target architecture but of general applicability throughout that architecture.

This system is designed to process a hardware description of a machine with respect to the above rules and performs one or more of the following tasks:

- A rule violation is located and reported to the designer.
- A potential rule violation is located and reported to the designer.
- Simulation data inputs designed to detect a potential error are generated for subsequent use.

Which of the above actions is performed depends on the nature of the rule or assumption which has been violated or has potential for being violated.

The assumption we are making here is that a significant number of design errors result from failure to adhere to conventional design practice, failure to adhere to rules imposed on the design, or from inconsistencies between assumptions about the design behavior and the actual behavior. These errors are to be distinguished from specific errors which occur once and which are not violations of a general rule which is correctly implemented elsewhere in the description. For example, failure to test for overflow after a particular add operation is a rule violation if the test for overflow is expected to occur after adds. Performing the add when it should have been a multiply is most often a specific error rather than the violation of a general rule.

The decision to look at some of the errors as rule violations allows us to partition the problem of searching for design errors into three parts. They are:

- Testing for incorrect behavior, which can be done by formal verification against a behavioral specification or by simulation.
- Testing for failure to adhere to design rules and assumptions, which can be done by formal verification against rules.
- Testing for failure to adhere to good design practice (e.g. timing violations) which can be tested for separately, either by formal verification or simulation.

We have focused here on the second and third parts of the problem. There is evidence that errors which can be categorized into rule violations occur in significant numbers. A cursory look at the design error survey conducted previously reveals that as many as 50% of the reported errors fall into this class. Most of the concurrent-type errors seem to be rule violations.

The specific work we have done under this contract is

- to show how to represent and process the design description, and
- to demonstrate rule violation detection capabilities.

The prototype system is intended to support a variety of error detection and prevention strategies. These include

1. taking many of the tedious and hence error-prone tasks of design out of the hands of humans;
2. building a large amount of knowledge about 'standard design practice' into the system;
3. making it easy to add knowledge about design-specific design practices;
4. providing a single semantically rich representation of the target specification, the design itself, and library components; and
5. providing a set of analysis tools that can be run either at the discretion of the system or the user at any time during the design process.

A short overview of relevant aspects of the system will serve both to explain the techniques involved and to review its current status.

At the highest level, the system can be classed as a *mixed planning and execution system*. Such a system is capable of formulating plans, in this case for design activities. It is not a pure planning system, however, because it mixes plan construction and execution. That is to say, the plans are constructed to cover future events up to a certain point, and then executed. The process of execution is really the expansion of the abstract plans into more detailed plans, until finally the expansion results in the actual execution of procedures that change the state of the design.

Because plans are made explicit in this system, and because all the lowest-level plans are accessible to the planner, it is possible for the planner to avoid *a priori* some kinds of design errors by establishing a standard of design practices implicit in the abstract task descriptions. For example, the task description frames can be set up to force the use of clocked registers in every combinational loop.

The planning system and frames are still under construction under other funding. A mechanism for plan retraction has been designed and is the means by which a plan can be aborted during its expansion because because of a violation. The mechanism used is called a *demon*, which describes a construct which waits for some condition to become true, and then executes an appropriate action. Demons are created during plan expansion; they are primarily used to monitor the various budgeted quantities. However, demons can be used to detect certain kinds of rule violations: for example, excessive fanout of a gate, or timing margins that are too small. The important point about such demons is that once they are set up, they can detect errors whenever they occur, no matter how remote the setup and error are.

Design data is represented in the following manner:

The Design data Structure (DDS) [Knapp 83a] is a multilevel design representation for use in the USC expert synthesis system [Knapp 83b]. It partitions design information into "subspaces" so that there are no implicit relationships between the objects of one subspace and the objects of another. One of the reasons for doing this is that the structural, timing, behavior and physical characteristics of a design do not decompose into isomorphic hierarchies. Therefore, the USC expert synthesis system uses four hierarchical design subspaces.

- The **data flow behavior subspace** represents the logical behavior of the target system. Behavior is specified by means of a single-assignment language, and internally represented as a graph, like a data flow graph in every respect except that arcs and nodes must be subscripted if they are to be used more than once, which is helpful when loops must be unwound. At the top level of the behavioral hierarchy a single node represents the behavior of the target machine; it is recursively subdivided into ever smaller nodes, until the nodes represent primitive functions.

- **The structural subspace** is analogous to a schematic diagram. It is a recursively defined hierarchy of "modules", which represent logical structures, and "carriers", which contain values.

- Physical properties are recorded in the **physical subspace**, whose primitive elements are "blocks" and "nets", and in which such things as power dissipation and size are kept. Thus the block "Z-80" might be recorded as being 2" X 0.6", with two watts of power dissipation. Layout information is contained in this subspace.

- The **timing and control subspace** is organized as a hierarchy of time "ranges", each of which is a partially ordered set of "points". Each range may have a physical (real) time length and a relational operator associated with it; for example, the range "interrupt-acknowledge-cycle" might have a time of 1 microsecond, implying the interrupt acknowledge cycle is less than

one microsecond long. This information may take the form of a simple fact, a tentative estimate, or a requirement. The timing view encompasses timing diagrams, timing constraints, and control-flow graphs in a single representation.

There are two classes of relations relevant to these subspaces, interspace and intraspace relations.

- Intraspace "links" are relations between components *inside* a subspace. There are **connected-to** and **composed-of** relations.

    1. **connected-to links**:Which usually connects two elements of the same hierarchy and makes them interdependent in one way or another.

    2. **composed-of links**: Which relates an element to its components which are at a lower level in the hierarchy.

- The four subspaces are algebraically related to one another by five types of interspace "bindings", or many-to-many relations. No other relations, implicit or explicit, exist between the objects of one subspace and the objects of any other. These binding types are

    1. **value to carrier to range**: this three-way binding states that a given value is to be found on a given carrier (i.e. a structural element capable of carrying a value) over the given time range.

    2. **node to module to range**: this binding states that a behavioral node is implemented by a particular module, and that the module is performing that function over the given time range.

    3. **block to module**: this binds a block of the static subspace to a module of the structural subspace.

    4. **carrier to block**: this binds a carrier of the structural subspace to a block of the physical subspace (presumably a register or memory unit).

    5. **carrier to net**: this binds a carrier to a physical conductor, for example a data bus.

Figure 1 shows an example of the four subspaces and bindings. Notice the non-isomorphism of the four hierarchies. The bindings are shown by the arcs connecting the hierarchies. For example, the arc connecting the **add** in the data flow behavior indicates when in the control and timing subspace the **add** is performed and that the ALU (in the structural subspace) performs the **add**.

The use of a semantically rich common representation gives the following advantages.

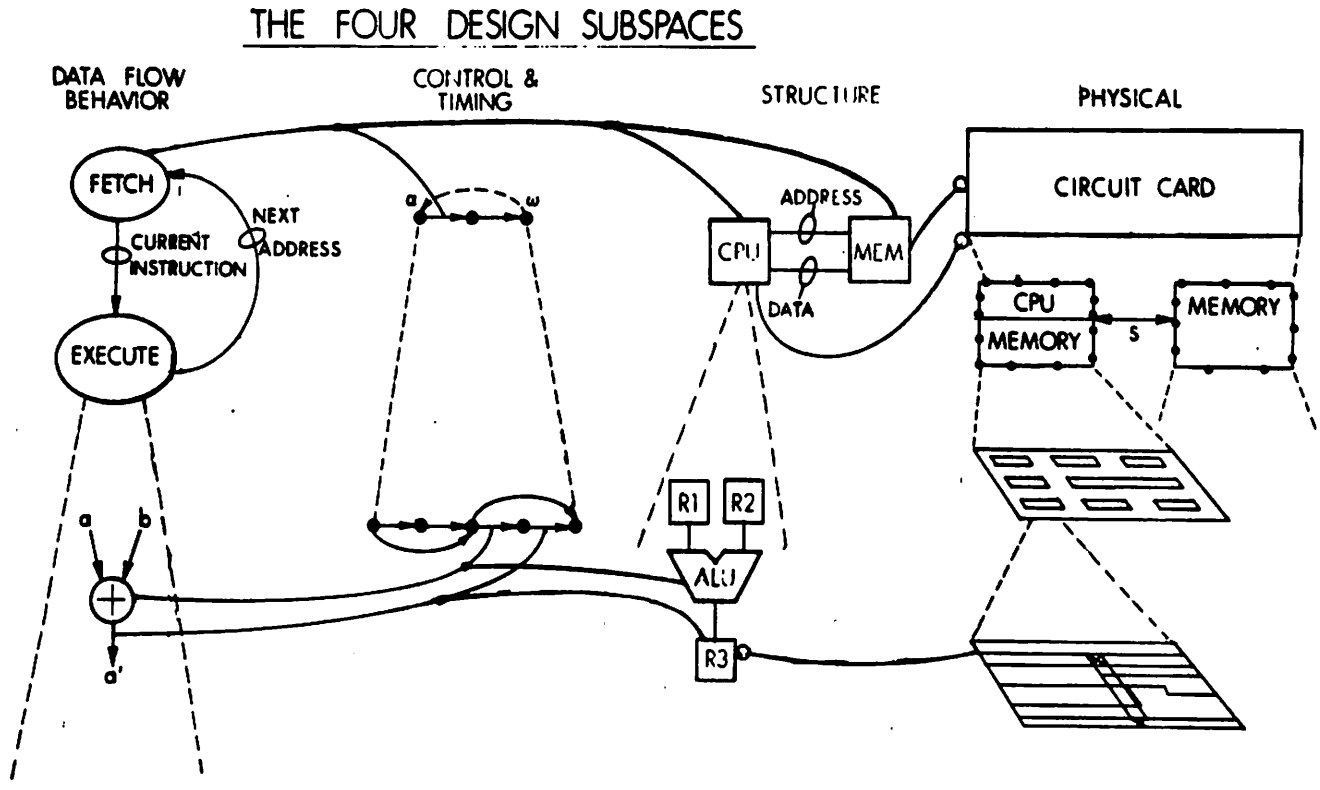


Figure 1: The Four Design Subspaces

1. Because the target specification and the current design are represented in the same way, comparing the behavior of the two is the same as proving the equivalence of two data flow graphs and two control flow graphs. Unfortunately this task is in general very difficult; luckily the system can make note of which construct corresponds to which, and what its reasoning was when it created the correspondence.

2. By making the relationship between an operation or value and the structural element in which it is performed or stored or transferred, explicit, errors such as

   a. value and operation collisions,

   b. use of dead values, and

   c. inappropriate or incomplete allocation.

becomes relatively easy to prevent and/or detect. The key to the use of this data structure is the way in which designs can be represented. A graphic interface (design editor) called AGIS has been implemented, which can be used in either stand-alone graphics mode, or as a syntactic and semantic checker of DDS descriptions.

The system plans tasks such as resource allocation in a straightforward way. Such tasks

directly affect the design, although their results may be tentative and hence easily retractable. Analysis tools, on the other hand, are used more to influence the decisions taken by the planner and by the user.

One such tool, the V collision detector, has been implemented as a stand-alone program. It is described in the next section.

4 Static Error Checking Using the Designer's Apprentice

In order to illustrate the use of the expert system design representation for error checking, one particular class of errors was chosen as an example, the collision.

4.1 Collision Detection Using V

A *collision* is a type of design error that is relatively easy to detect in the DDS [Knapp 83a]. Collisions occur when single hardware modules are assigned to tasks that overlap one another in time. Such errors occur in two basic forms:

1. Value collision, where a data value thought to be in a particular register (or bus or memory) is forgotten because another value is overwritten into the register.

2. Operation collision, where a data-transforming module is excited to perform two (or more) transformations at once, and where the module is not known to be capable of such simultaneous operation.

Collisions may be easy to detect under some circumstances, particularly in those cases where there is little parallelism. In other cases, where there is a good deal of parallelism and the parallel streams have conditionals, it is not always clear that there either is or is not a collision in some module.

Simulation is a classical way to deal with this problem. However, as the number of conditionals grows, simulation becomes more and more unwieldy. The V collision detector can be regarded as either a partial replacement for simulation, or as an augmentation of the power of simulation.

1. If V reports that there are guaranteed to be no collisions in a design, then there are no collisions. Simulation can be used to check execution paths as if they were disjoint, because paths do not interfere with one another in unplanned ways.

2. If V reports that there are potential collisions, then the design may still be correct; the simulation and/or symbolic simulation can concentrate on the reported error possibilities and ignore other behavior.

At the moment, V is configured as a stand-alone program with a preprocessor that

enables its use with the Agis DDS editor [Knapp 84a]. It is documented in [Knapp 84b]. It runs under Berkeley 4.2 BSD Unix, is written in Franz Lisp [Foderaro 80] and is about 900 lines long. The uncompiled code occupies about 26 Kbytes of space.

The remainder of this section gives a brief overview of the relevant aspects of the DDS, then discusses the algorithm used in V, and finally gives a discussion of 'false positives'. Appendix 2 gives arguments on the correctness of the algorithm and its complexity.

4.2 The Design Data Structure: Relevant Aspects

The DDS is based on a principle of separation of functional, structural, timing and physical information. Each of these classes is relegated to one subspace of the global design space [Director 81]. The intent of the partitioning is that aspects of the design which can be freely manipulated without necessarily considering other aspects be represented in such a way that this freedom is reflected in the design representation. We call this property the *orthogonality* of design subspaces.

Of the four orthogonal design subspaces, the V program considers only three: the dataflow subspace, the control and timing subspace, and the structural or schematic subspace.

The way in which objects in these subspaces are related is expressed by *bindings* across the subspaces. There are two types of bindings relevant to V; *nmr*-bindings and *vcr*-bindings. An nmr-binding is a relation that expresses the connection between a dataflow node, a schematic component, and a time range; e.g. between an addition, an ALU which performs the addition, and a microcycle, during which the addition is performed. A vcr-binding represents the connection between a dataflow value, a schematic carrier, and a time range, e.g. an operand found on a bus during a microcycle. The carrier of a vcr-binding, along with a module, can represent a register; a carrier is essentially anything to which a value can be bound.

The place of V in this scheme of things can now be described. V simply checks the sets of vcr- and nmr-bindings to see that there are no cases in which different dataflow objects are bound to a single schematic module or carrier during time ranges which may overlap.

Because hardware designs tend to be less structured than software programs, these structures are not always properly nested; V merely makes the assumption that the timing graph is (1) directed, and (2) acyclic, with no other restrictions of well-formedness or nesting. This implies that V will not report an ill-formed construct, e.g. one which has multiple global sources of time constraints (and therefore is not a lattice); such structures are questionable in software design, but seem to be the rule in many kinds of hardware design. Rather than restrict the hardware designer to a software-like structuring discipline, V simply assumes that the designer 'wanted it that way' and reports potential collisions on that basis.

4.3 The Algorithm[2]

The V program consists of three main blocks, organized serially. These three blocks perform the following tasks:

1. The input processor reads in text file representations of designs and constructs preliminary data structures.

2. The error processor performs the actual collision detection.

3. The output processor reformats and outputs the results of the collision detection operation.

The input and output processors' data structures are:

1. A timing graph, consisting of a list of nodes and a list of arcs linking those nodes (input).

2. A list of modules, with the operation and time range bindings of the modules (input).

3. A list of colliding binding pairs (output).

These processors are rudimentary, and they will not be discussed further. The error processor operates on the timing graph and the module list to produce the list of colliding binding pairs. In what follows, the key to understanding the procedure is to bear in mind that the timing graph is a directed acyclic graph (DAG). The procedure given would not work for a more general type of graph. The procedure is broadly as follows:

1. Traverse the graph in a forward direction, assigning unique sequence numbers to arcs and nodes such that the total ordering of sequence numbers is compatible[3] with the partial ordering created by the DAG. At the same time, create the 'crushed' timing graph, i.e. the timing graph with all simple (unbranched) chains of arcs replaced by single arcs.

2. During traversal, collect sets of predecessor arcs {A} for each arc.

3. Form the set {C1}, which is the set of arc pairs (X,Y) such that X neither precedes nor succeeds Y. This is the set of *potentially* conflicting arc pairs.

4. Form the set {C2}, the set of *actually* conflicting arc pairs, by examining sets

[2]The casual reader is invited to skip this section if the details of the algorithm are not of interest.

[3]Compatible in this context means that the morphism is order-preserving so that any node or arc does not have a sequence number less (greater) than that of any of its descendents (predecessors).

of disjoint paths by which the arcs X and Y could be reached simultaneously. Do not add pairs that have absolute times attached to them such that the pair can be guaranteed not to overlap.

5. Examine the set of bound ranges {R} of each module M. If two bound ranges, R_1 and R_2, both of M, have an arc pair (X,Y) ϵ {C2} in common, then the bindings are in collision. Add the pair (R_1,R_2) to the set {C3}, which is the set of collisions to be reported.

Figure 2 contains a timing graph with conflicting and non-conflicting range pairs.

4.3.1 Forward Traversal

Traversing the graph in order and assignment of sequence numbers are done by the following algorithm:

1. The list of simple source nodes {C} is formed. It consists of all nodes with in-degree zero and out-degree one.

2. The list of unresolved fork source nodes {U} is formed. It consists of all nodes with in-degree zero and out-degree greater than one.

3. Repeat the following steps until both {C} and {U} are empty:

   a. If {C} is not empty, take the first element from it and label its out-arc with the current sequence number. Remove the node from {C} and traverse the out-arc.

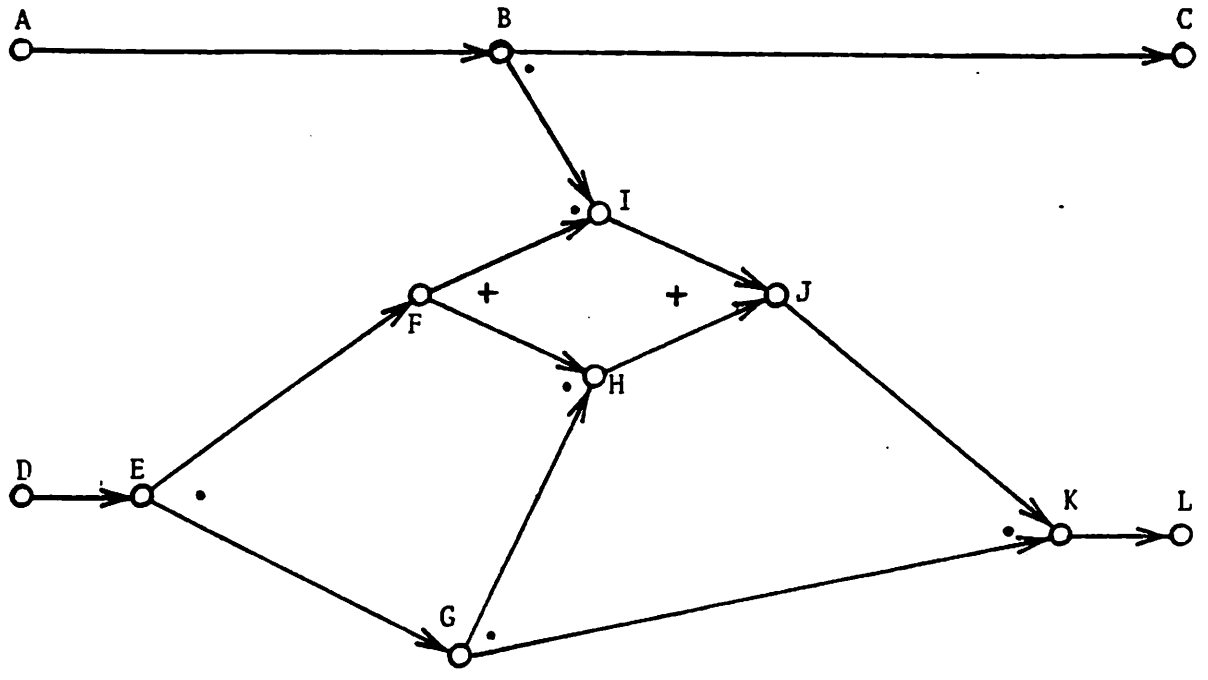
   b. If {C} is empty but {U} is not empty, choose an unnumbered out-arc of the first element of {U} and number it. Traverse the arc, and if there are no more unnumbered out-arcs of the first element of {U}, remove that element from {U}.

   In traversing an arc, the treatment of the sink node of the arc is determined by its degree and the number of numbered in-arcs it has.

   a. If it has in-degree greater than one, and not all its in-arcs are numbered, it is ignored.

   b. If all of its in-arcs are numbered, it is appended to either {U} or {C}, depending on the number of out-arcs it has. If its out-degree is one, it is appended to {C}; if its out-degree is greater than one, it is appended to {U}. If its out-degree is zero it is ignored.

Because no join node is considered for out-traversal until all of its in-arcs are numbered, and because no fork node is out-traversed until all preceding forks and other nodes have been exhausted, the order of numbering and hence of appending to the lists {C} and {U} is an order-preserving morphism of the partial order imposed by the DAG.



(EF EG) $\in$ C2

(FH GH) $\in$ C2

(EF GK) $\in$ C2

(HJ IJ) $\in$ C2

(FI FH) $\notin$ C2

(AB EF) $\in$ C2

Figure 2: Conflicting and Unconflicting Range Pairs.

Because no direct predecessor of an arc can be unnumbered when that arc is numbered, the predecessor set can be formed and is complete.

4.3.2 Building the Successor Sets

Each predecessor of the current range has a successor set to which the current range is added.

4.3.3 Form the Set of Potentially Conflicting Range Pairs

The set {C1} of potentially conflicting range pairs is formed by subtracting the set of predecessors and successors of each range R from the entire set of ranges. The set that remains is the set with which R can be in conflict. This set of range pairs is the set of ranges that overlap in time, but because of the semantics of the or-forks that may be included in the graph, the ranges may have the same temporal coordinates, but they may actually exclude one another by forming two branches of a conditional. For that reason, it is necessary to further refine {C1}.

4.3.4 Forming the Actual Conflict Sets

The actual conflict set {C2} is formed from the pairs (R_1,R_2) that form {C1}. Each pair is checked and either added to {C2} or not, depending on the results of two tests:

1. If it can be demonstrated that the latest time R_1 can end is earlier than the earliest time R_2 can begin, or vice versa, then the pair is not added to C2 and the second test (below) is not performed.

2. Otherwise, it is necessary to determine if two disjoint paths can be backtraced from the source nodes of R_1 and R_2 to a single common and-fork. If this is the case, the two paths correspond to two paths back to a common parallel-begin statement, and the pair (R_1,R_2) is added to {C2}. Under some circumstances, this test has an ambiguous result; if that happens add the pair to {C2}.

The absolute time exclusion test is easily implemented by attaching to each node a upper-bound/lower-bound pair, and then comparing the upper and lower bounds of R_1 and R_2.

The disjoint-path test is implemented by an algorithm quite similar to the forward-traversal algorithm discussed above. The following data structures are used:

1. {C1}: This is the {C1} constructed in the previous steps, consisting of pairs of ranges (R_1,R_2) such that neither is a predecessor of the other.

2. {C2}: This is the subset of {C1} that contains the pairs (R_1,R_2) such that R_1 and R_2 cannot be shown to exclude one another. There are two conditions of membership in {C2}, both of which are discussed above.

3. {C}: This is a set of simple nodes.

4. {U}: This is a set of join nodes.

5. {W}: This is a set of fork nodes.

6. {V}: This is a set of nodes which have no predecessor arcs, i.e. global source nodes.

Essentially, the disjoint-path test is a graph traversal problem. In a trivial case, R_1 and R_2 have a common source node. If that node is an AND-fork, the two are in conflict; if it is an OR-node, they exclude one another and are not in conflict. In a less trivial case, the sources of R_1 and R_2 are marked with an 'a' and a 'b' respectively, and added to a 'wavefront' list. The arcs preceding nodes in the 'wavefront' can then be marked themselves, and so on, until one of three conditions applies:

1. there is only one node in the wavefront,

2. an AND-fork node is encountered with successor arcs marked both 'a' and 'b', or

3. the wavefront cannot be propagated because its members have no predecessors.

The first case means that the pair (R_1,R_2) should not be put into {C2}, while the other cases mean that it should.

The detailed algorithm will now be given. The procedure is complicated by the presence of fork and join nodes. All of the reachable successors of a fork should be marked before any of its predecessors; in general not all the successors of a fork are reachable by backward traversal. We must therefore traverse backward in the reverse order of the precedence relationships, i.e. by height.

For these reasons, the wavefront nodes are put into four sets, {C}, {U}, {W}, and {V}. {C} is the set of simple nodes. These will always be traversed in preference to the others. When a member of {C} is traversed, it is removed from {C} and never considered again. {U} is the set of join nodes; one of these will be traversed only if {C} is empty. Traversing a member of {U} is done by marking one of its predecessors and adding it to the wavefront. If all of its predecessors are marked, it is removed from {U}. {W} is the set of fork nodes; its members are only traversed if {C} and {U} are both empty, and furthermore, it is kept in sorted order, so that no element of {W} can be traversed before any of its successors. When a member of {W} is traversed, it is removed from {W}. If at any time there is only a single member of {W} in the wavefront, the procedure is terminated, because all longer paths to R_1 and R_2 must pass through that node and therefore are not disjoint.

Finally, {V} contains nodes which cannot be traversed because they have no predecessors. If there is more than one element of {V}, R_1 and R_2 are in conflict. This is the case because either

1. R_1 and R_2 have no common source node, and hence their relative timing cannot be determined.

2. R_1 and R_2 have at least one common source node and under some OR-fork conditions have different source nodes.

At each encountering of a fork, its out-arcs are tested. If it is an OR-fork and it has at least one 'a' and at least one 'b', then its in-arc is marked with an 'ab', signifying that either R_1 or R_2 could descend from it. Similarly, if any out-arc is marked 'ab', then its in-arc is marked 'ab'. If, on the other hand, the fork is an AND-fork and it has two 'ab' out-arcs, or two differently marked out-arcs, then R_1 and R_2 are potentially in conflict and the procedure is terminated.

The detailed algorithm is given in Appendix 2.

An argument for the correctness of this algorithm, and another on its complexity, are presented in Appendix 2. Briefly, the correctness of the algorithm hinges on no fork node ever being traversed twice; if this is the case, and all predecessor nodes (back to a single common source) are visited, the tags of the out-arcs of any fork must be up-to-date at the time the fork is traversed. Because the tags are up-to-date, they provide a conclusive test that either R_1 and R_2 are in conflict or that that particular fork need never be considered again.

4.4 False Positives and Incomplete Designs

A 'false positive' report from V can mean one of two things. Either the absolute timing information is inadequate, and two ranges seem to be concurrent when they are not, or the timing graph has more than one source node.

If a dearth of absolute timing information is the cause of a false positive, it is as if an arc were missing from the graph; an ordering that really does exist is not expressed.

If there is more than one global source node, that too is expressive of something missing: either two parts of the design are not synchronized (which may not be a problem) or their synchronization has not been expressed in the timing diagram.

In both of these cases, the false positive result really means that the design is incomplete; it may not work under the given constraints. This is not a bug, it is a feature: it allows the program to be run on incomplete designs, with reportage on the incompleteness as well as the correctness of the design.

4.5 Summary

A description of the function and structure of a program called V has been given. The program detects certain kinds of design errors and incompletenesses, all of which are characterized as 'collisions' of one sort or another. A 'collision' is defined as simultaneous dual or multiple usage of a hardware resource, where each usage by itself may be correct, but the simultaneous usage constitutes a violation of the ground rules of the resource. Collisions are easily detected using V and the DDS [Knapp 83b] in at worst $O(R^3)$ time, where R is the cardinality of the set of explicitly defined time ranges between events.

5 Conclusions and Recommendations

The following conclusions have been arrived at:

- Design errors fall into types
- Many design errors are violations of general rules
- Any formal verification technique will guarantee correctness only if the design is compared to an abstract specification, or if the errors detected are violations of general rules which have been previously stated. Furthermore, most formal specification techniques are complicated, and the specifications are lengthy.
- Even if a higher-level specification were provided, oversimplification of the problem could lead to a false sense of confidence about lack of errors.
- Numeric simulation is guaranteed to detect errors only if the test cases are carefully constructed to detect each class of errors, leading to a combinatoric explosion.
- Syntactic (static) analysis can be used to detect some errors if the design data is properly represented.
- Symbolic processing of design descriptions can be guaranteed to reveal general concurrency violations, if the design data is structured properly with explicit control and data flow.
- Current HDL's tend to obscure control flow, data flow or both. Better hardware description techniques might themselves alleviate some errors.
- Deterministic error coverage figures are impossible to obtain due to the variety of errors and the situations under which they occur. Error coverage figures will have to be estimates based on the probabilities of error

occurrence, and the related situations. Therefore more statistical data would have to be gathered before probabilities of error detection could be computed.

- Many errors involve asynchronous timing, exceptions due to hardware failures, parallelism or concurrency. Current formal verification techniques and simulators only cover limited, special cases of those techniques.

- Many errors are difficult to detect due to the combinatoric explosion of possible common states between different hardware processes.

- Symbolic simulation primarily detects logical design errors (errors in function, not control flow), which can be detected as easily by numeric simulation.

These conclusions have led us to make a number of recommendations:

- Abstract system specification techniques should be investigated and formal models defined. Work in using natural language for system specifications is underway at USC. Concurrency and asynchrony are being studied.

- Design descriptions should be translated into an internal format which makes data flow and control flow explicit.

- A compilation of design practice violations and design rules should be undertaken.

- Correctness by construction techniques (manual or aided by a computer) should be used to guarantee correct concurrent, asynchronous behavior. Such "structured" hardware designs would have a small cost or performance penalty in many cases and should be implemented carefully.

- More statistics on error probabilities should be gathered in order to determine how to target test cases.

References

[Allen 76] Allen,F. and Cocke, J.
A Program Data Flow Analysis Procedure.
Communications of the ACM 19(3):137-147, March, 1976.

[Andrews 80] Andrews M.
Principles of Firmware Engineering in Microprogram Control.
Computer Science Press, 1980.

[Dasgupta 76] Dasgupta, S. and Tartar, J.
The identification of maximal parallelism in straight-line microprograms.
IEEE Transactions on Computers C-25(10):986-992, October, 1976.

[Director 81] S. W. Director, A. C. Parker, D. P. Siewiorek, D. E. Thomas.
A Design Methodology and Computer Aids for Digital VLSI Systems.
IEEE Transactions on Circuits and Systems CAS-28:634-645, July, 1981.

[Feldman 79] Feldman, J. A.
High Level Programming for Distributed Computing.
CACM 22(6), June, 1979.

[Foderaro 80] Foderaro, J.
Franz Lisp Manual.
Computer Science Division, EECS, University of California Berkeley, California.
1980

[Goodenough 75] Goodenough, J. B. and Gerhart, S. L.
Toward a Theory of Test Data Selection.
IEEE Transactions on Software Engineering SE-1(2), June, 1975.

[Hitchcock 82] Hitchcock, R. B., et. al.
Timing Analysis of Computer Hardware.
IBM Journal of Research and Development 26(1):100-105, Jan., 1982.

[Knapp 83a] Knapp, D. and Parker, A.
A Data Structure for VLSI Synthesis and Verification.
Technical Report, Digital Integrated Systems Center, Dept. of EE-Systems, University of Southern California, October, 1983.

[Knapp 83b] Knapp, D. ,Granacki, J. and Parker, A. C.
An Expert Synthesis System.
In *Proceedings of the International Conference on Computer Aided——— Design(ICCAD)*, pages 419-424. ACM-IEEE, September, 1983.

[Knapp 84a] Knapp, D.
The Agis Data Structure Editor.
USC DA Group, available on-line.
1984

[Knapp 84b] Knapp, D.
The V Collision Detector.
USC DA Group, available on-line.
1984

[McFarland 78] McFarland,M.
The Value Trace: A Data Base for Automated Digital Design.
Master's thesis, Dept. of Electrical Engineering, Carnegie-Mellon University, Pittsburgh, Pa., December, 1978.

[McFarland 83] McFarland, M. and Parker, A.
An Abstract Model of Behavior for Hardware Description.
IEEE Transactions on Computers C-32(7):621-637, July, 1983.

[Milner 82] Milner, R.
A finite delay operator in Synchronous CCS.
Technical Report, Dept. of Computer Science, University of Edinburgh, May, 1982.

[Moszkowski 82] Moszkowski, B.
A temporal logic for multi-level reasoning about hardware.
Technical Report STAN-CS-82-952, Department of Computer Science, Stanford University, December, 1982.

[Nagle 81] Nagle, A. and Parker, A.
Algorithms for Multiple-Criterion Design of Microprogrammed Control Hardware.
In *Design Automation Conference Proceedings no. 18*, pages 486-493. ACM SIGDA, IEEE Computer Society-DATC, June, 1981.

[Park 84] Park N. and Parker, A.
Synthesis of Optimal Clocking Scheme for Digital Systems.
Technical Report DISC/84-1, Dept. of EE-Systems, University of Southern California, May, 1984.

[Parker 83] Parker, A.
Simulation Effectiveness.
Technical Report DISC/83-2, Dept. of EE-Systems, University of Southern California, April, 1983.

[Parker 84] Parker, A., Kurdahi, F. and Mlinar, M.
A General Methodology for Synthesis and Verification of Register Transfer designs.
In *Proceedings of the 21st Design Automation Conference*. ACM SIGDA, IEEE Computer Society, June, 1984.

[Pitchumani 82] Pitchumani, V. and Stabler, E.P.
A Formal Method for Computer Design Verification.
In *Proceedings of the 19th Design Automation Conference*, pages 809-814. ACM SIGDA and IEEE Computer Society DATC, June, 1982.

[Pitchumani 84] Pitchumani, V.
An Assertion-oriented Method for Verification of Parallelism in Hardware.
In *IEEE Proceedings of the ICCD*, pages 462-467. October, 1984.

[Shaw 78] Shaw, A.
Software description with flow expression.
IEEE Transactions on Software Engineering SE-4(3):242-254, May, 1978.

[Tran 82] Tran, A., Forsberg, R. and Lee, J.
A VLSI design verification strategy.
Technical Report 4, IBM, July, 1982.

Appendix 1

SIMULATION EFFECTIVENESS LITERATURE SURVEY

F. Kurdahi, N. Park and A. Parker

30 October 1984

Table of Contents

1. Introduction

This report summarizes published research relevant to the problem of simulation effectiveness. Virtually no research has been reported on test generation for simulation done to determine logical correctness. However, design errors are covered in some reliability models, and there are some analogies to software testing. With the relationship to software in mind, previous research in the areas of hardware simulation, hardware verification, software verification and software testing was examined. A computer search was conducted by searching from 1976 through 1981 on the following topics:

- *Hardware simulation*
- *Hardware formal specification*
- *Hardware timing errors*
- *Hardware verification*
- *Hardware design errors*
- *Hardware validation*
- *Hardware logical correctness*
- *Concurrent hardware*
- *Software verification*
- *Software test data selection*
- *Software correctness*
- *Software errors*

Papers from each of these research areas were surveyed. Some of the most relevant publications are reviewed here. Some relevant publications which have been repeatedly cited have been omitted here.

2. Hardware Simulation

[VanOost 81] presents some difficulties of using multi-programming languages in simulating a multi-processor system. Structured multi-programming languages (e.g., P. Brinch Hansen's Concurrent PASCAL) are very useful to define and describe a system in a structured way (hierarchical description), but they are only useful for simulation if a certain level of abstraction can be preserved.

In fact, if we use a structured programming language to design a hardware system, there will be significant amount of redundancy in the designed system. This is due to the fact that the resulting system must be structured and therefore resource sharing is significantly limited. Also, in simulating a hardware system using a structured programming language, there will be too much inefficiency if we simulate the system at a low level (e.g. gate level). This is due to the overhead of linking (or translating) between levels.

In conclusion, a structured programming language is not so useful in either designing an efficient (cost vs. performance) system or in simulating a system at a low level. However, if we can automatically translate the program written in such a language into an efficient simulation program or internal description, then it will be very useful since structured programming languages or description are much easier to write and understand than most efficient simulation languages.

[Huebner 79] describes automatic generation of test cases for telephone switching systems. The test cases are generated by translating the requirements specification into stimulus/response sequences. Tests of invalid stimulus sequences are constructed by modifying valid stimuli to reflect required responses to invalid stimuli. These are also initial condition stimuli to force the system into an initial state, and error recovery stimuli used to return the system to a designated state. This paper proposes this software, rather than describing complete results.

3. Hardware Formal Specification

[Lauer 80] discusses a method of designing and analyzing systems based on a linguistic rather than a graph-theoretical descriptive formalism. A high-level descriptive notation which translates down into Habermann's path expressions is provided as a tractable design tool. Net theory (especially Petri-Nets using vector firing sequences) is used for a formal verification tool. (This is the origin of behavioral semantics.)

[Foulk 80a] describes the differences between H/W description and S/W description as follows: "Description of hardware is different from that of software, since it does not have flexible data structures, coroutines, heaps, stacks, and recursion (you cannot design a microprocessor, one of whose components is an identical microprocessor). However, this does not mean that one could not design hardware to execute a recursive function. Only the description of the hardware is not in itself recursive."

With his classification, parallel systems can be categorized into the following four types:

1. Type 0 : Purely sequential control. Most application software falls into this category.

2. Type 1 : Time sharing control. Several processes are controlled by a time-shared operating system.

3. Type 2 : Centralized control with multiple processors. Parallel nodes of a control graph can be implemented with this type of controller.

4. Type 3 : Multiple independent controllers and processors. A distributed computer system is a Type 3 system.

In fact, we can see all four types in a computer system. The authors use the G language which is a combination of graphs (a data flow graph and a control graph) and a concurrent PASCAL-like language to describe any type of parallel system. The graphs are based on LOGOS.

The focus of this paper is just on the G language. It uses some features of concurrent

PASCAL to represent concurrent hardware processes. It also has some graphical features so as to construct data-flow and control-flow graphics easily. However, it does not represent asynchronous processes and timing. The representation is less complete than our own expert system data structure.

[Foulk 80b] claims that the requirements of an ideal hardware description/design language are:

1. It should have universal acceptance by human designers.
2. It should allow the designer to go into as much detail as required about any aspect of his design.
3. It should reflect both the structure and the function of the target system.
4. It should have separate control flow and data flow.

The AIDS system introduced in his paper has the following characteristics:

1. Hierarchical : Detailed description is included by virtue of hierarchical construction of description.
2. System level description consists of a data flow graph and a control graph. As a tool for automatic generation of the graphs, a graphic language, G, is used.
3. An HDL is used for the RT-level description.
4. For signal (circuit) level description, the RT-level description is decomposed in a hierarchical manner.

The AIDS system is purely hierarchical. Therefore, the design process is very structured and it is easy to analyze, verify, and synthesize. However, since there is no explicit relationship between levels of description, there is little chance of exploring all the parallelism of the target system. This also makes global transformation and/or optimization difficult.

[Brand 83] models a network protocol as a set of communicating finite state machines. Three basic approaches to the modeling of network protocols are compared as shown below.

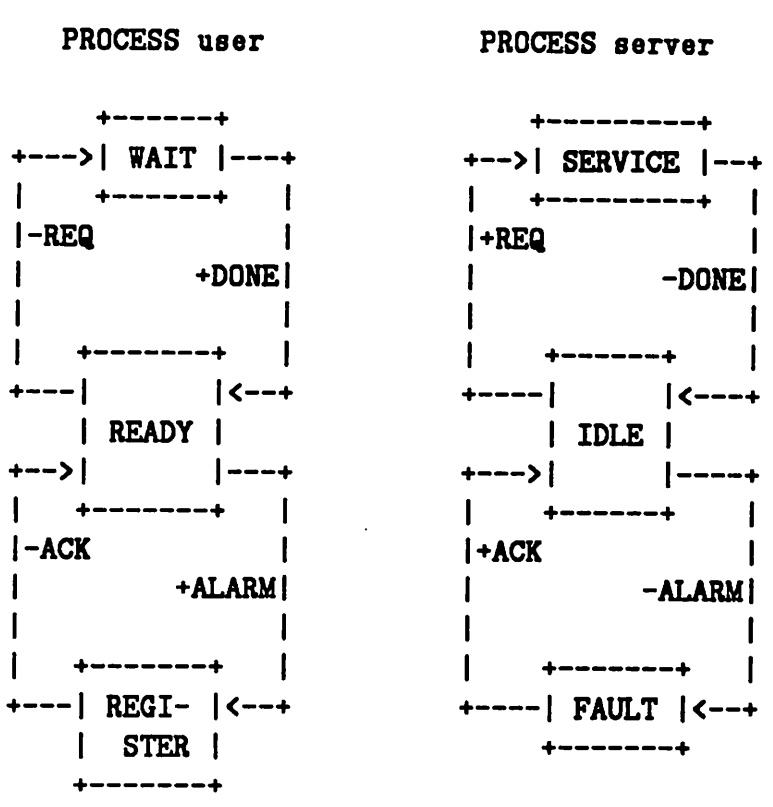
	Parallel programs	Petri-net	Finite-state machine
	most general	---------->	least general
Merits	can specify all protocols and most of their properties	more easily analyzable and decidable than parallel programs	all describable protocols are decidable and easy to analyze
Demerits	undecidability	less general	only certain protocols can described
Application	use human assistance or do not use all the generality available	add some features of parallel program model. However, it makes analysis harder.	

In this paper, the authors use the f.s.m. model. (Obviously, a protocol which allows an arbitrary number of messages in transit cannot be described.)

In this model,

1. Each process is a finite-state machine.
2. Each pair of communicating processes is connected by a bidirectional FIFO channel.
3. A complicated channel can be separately represented by another f.s.m.

Example



In this example, each box represents a state of a process. A directed edge between two boxes represents a state transition. A transition is triggered by either sending a message (a message with + sign) or receiving a message (a message with - sign).

For example, suppose that "user" is in the READY state and "server" is in the IDLE state. "user" enters the WAIT state by sending "REQ" to server and "server" enters the SERVICE state by receiving "REQ" from user.

Actually, there are two essential requirements for the processes to run in a more concurrent fashion :

1. A buffer area to store messages in each channel.
2. Sending or receiving a stream of messages instead of a single message from one side in advance to the responses from the other side.

To represent the above two cases in finite-state machines, we need multiple edges of a direction between states of a process.

The two major problems attacked in the above paper are:

1. How to synthesize (draw) the finite-state machines from a verbal design description.

2. How to verify the finite-state machines (correctness, dead-lock, etc.).

The approach to the problems suggested here can be summarized as follows:

1. Synthesis of finite-state machines: Synthesis starts by building a tree in which nodes are states (not necessarily distinct). The root is the starting (initial) state of the f.s.m. Children of a node are possible next states of the node. The designer first builds the tree and, from the tree, he/she synthesizes a finite-state machine.

2. Verifying correctness of the f.s.m. : Using reachability (transitively closure) between states, they check the correct sequence of message transfer.

This model is very straightforward and easy to analyze. However, they have some restrictions which prevent the model from being practical.

The first restriction is that they must have a error-free channel for message transfer.

Secondly, building a tree with proper depth is not trivial. (How deep the tree should be and, from the tree, how to synthesize a f.s.m. with optimal number of states.)

The other restriction of this model is that it can only model relatively simple protocols with a small amount of message transfers. This is due to the fact that building and analyzing a f.s.m. with more than 100 states is almost impossible.

Note that application of the f.s.m. model is limited to verifying the correctness of protocols between two systems. It cannot be used as a synthesis tool. Also, synthesizing a

protocol only does not have much meaning since its implementation is greatly dependent on the behavior and structure of the I/O processes of the communicating systems.

[Moszkowski 82] describes an extension of temporal logic which could be helpful to use in our model of concurrent behavior. The author deals with signal values at the beginning, end and throughout an interval. He defines signal stability, and talks about subintervals and length of intervals. He deals with dependency of signals on other signals, signal transitions, simple delay, simple repetition, and assignment of values during an interval. He does not deal with concurrency or concurrent process issues directly. Unfortunately, the work lacks sufficient depth so much more research would have to be done before it could be applied.

4. Hardware Timing Errors

[Hitchcock 82] proposes a block oriented automatic timing analysis algorithm to identify any logic with timing problems. The algorithm associates three parameters with each logic block: the block delay, the arrival time of the output (established by selecting the last input to propagate to the output), and the slack, which is the difference between the required AT (arrival time) and the actual AT. The algorithm also provides a statistical analysis of the block delay and other parameters. The complete TA (Timing Analysis) model consists of combinational logic, sequential logic and the clocks. By specifying two different parameters for a certain clock with respect to sequential and combinational circuits, the TA program can provide forward and backward analyses in one run. The program also allows the insertion of delay modifiers in the circuit. Hence, by using the slack values produced by the TA program, the designer can pinpoint timing problems in a hardware design.

5. Hardware Verification

[Rault ??] tries to specify the steps in digital circuit development in order to partition verification steps between each design phase. He also surveys the latest DA tools for different levels of verification. The paper identifies nine steps in a design process:

1. Drawing Requirements (functional and physical specs)
2. Checking specifications for consistency and completeness
3. Selecting a functional architecture
4. Selecting components for physical realization
5. Checking that integration constraints and initial specifications are met
6. Drawing masks
7. Analyzing masks and checking design rules
8. Manufacturing
9. Controlling quality

With these steps in mind, the authors subdivide hardware verification into five stages :

(A). Writing and verifying specifications: Writing specifications corresponds to abstract entities whose purpose is to define functions and characteristics of the product for both design and implementation. Verifying specifications is done in two steps :

- Informal ; i.e. for natural language specifications, this is a heuristic approach.
- Formal ; by using flow charts or state tables, for example.

(B). Validation of architecture : Involves the selection of circuit architecture and validates it using such tools as function simulators, consistency verifier and an assessment program.

(C). Verification of logical and electrical implementations :

The electrical verification requires the use of a general purpose, technology simulator

and linear and non-linear simulators, to analyze the AC, DC, and transient behaviors, sensitivity, noise, faults, optimization, etc.

Logic verification of the realization requires the use of a logic simulator. Timing verification is performed in order to uncover potential delay faults by using tools for fault propagation sensitizable paths, stochastic simulators, and structural analysis of circuit graphs.

(D). Layout verification : Involves

- Logic and electric verification at two levels ; analysis of topological structures and analysis of electrical and logic responses.
- Verification of "microlithographic constraints" (i.e. of design rules) by using DRC's.

(E). Verification of manufacturing data : Requires solving a statistical modeling problem; Monte Carlo methods can be used but are costly in computer time. Instead, another approach is available and uses two analyses. The first provides distribution profiles, the second provides the nominal values and the distributions of the electrical parameters of models.

[Hailpern 80] uses logical assertions to reason about classes of states :
Safety - bad things(e.g. deadlock) will not happen.
Liveness - the future occurrence of desired state is guaranteed.

"square"P - P is true for all states in the future.

"diamond"P - there is some state in which P is true in the future.

At P - assertion true at the beginning of program.

After P - assertion true at termination.

An invariant describes the safety properties of the medium.

[Pitchumani 82] presents a formal design verification method based on an inductive assertion technique as a possible alternative to logic simulation. The inductive assertion method requires assertions at the start and end of the program, and at least one within each loop, each assertion specifying relationships between the current values of variables. An automated system for verification will, therefore consist of two parts: a verification condition generator and a theorem prover. In order to apply this method, the authors formally define a register transfer language (similar to AHPL). For that purpose, they start by defining three axioms: the Null, the unconditional transfer and the conditional axioms. Then they define six rules governing the interaction between the axiomatic elements. Termination of the program loops is imposed by including an upper bound for time in the loop invariant. Once the language is defined, verification of a design specification can proceed in a way similar to software verification, as presented in the example treated in the paper.

[Umrigar 83] adds time to pre- and post-conditions used as assertions in proving hardware correctness through symbolic simulation. Since timing errors within a single process can be solved with timing verification methods, this paper does not contribute to our research directly.

6. Hardware Design Errors

No papers which best fit this category were discovered in the course of the literature survey.

7. Hardware Validation

[Tran 82] describes using a hardware prototype for exhaustive functional testing. The authors claim that exhaustive simulation is too costly and time-consuming. This way, they can also test the I/O because the hardware prototype can be embedded in an actual system.

8. Hardware Logical Correctness

No useful papers were discovered here except those which fit better into other categories. Relevant papers are [Cory 81], [Leinwand 82], [Pitchumani 81] and [McFarland 83].

9. Concurrent hardware

[VanMierop 79] uses distributed interacting processes to define communication networks, distributed data bases, multi-microprocessors, CPUs interacting with their peripherals and redundant computer systems.

He believes that parallelism, nondeterminism and timing dependencies make understanding the behavior of distributed systems very complicated. For this reason, he uses the Structured Model of Behavior (SMB), which is based on and is almost the same as UCLA's Graph Model of Behavior (GMB), to describe concurrent hardware processes in a structured way.

The implementation method in this thesis is the same as the GMB model implementation. The verification technique uses the symbolic execution method.

He found three limitations in automating his implementation and verification methods. They are : (i) an implementation only specifies the proper cycling of concurrent processes and their logical communication links, (ii) the symbolic execution can run forever and (iii) the symbolic execution can not prove correctness of some aspects of the hardware. Also, timing aspects in verification are completely ignored. Thus, as he clearly stated, his methods can not be applied to general design problems.

The SMB model

A system consists of processes interconnected with message-links. All the processes are activated in parallel after activation of the system.

A process consists a sequence of statements. It access remote (shared) variables using remote-assignment. comm-statement is used for message transfer. Each process has a process-counter as its own active statement counter (similar to program counter).

A local-assignment assigns a value to a local (internal to the process) variable.

An if-statement is used for conditional execution.

A while-statement is used for conditional iteration of a loop.

A remote-assignment is used to access a remote (external) variable. It is either SEND or RECEIVE.

A comm-statement is used to define the behavior of a complex message link. If a comm-statement is activated, one of the alternatives of remote-assignments in the comm-statement is selected and executed nondeterministically.

In implementing a system, they only specify the system down to block diagram level, which is in between the abstract behavioral level and register-transfer level. Needless to say, as the author stated, this is due to the lack of the capability of his model (SMB) to describe low level details of hardware. The missing or insufficient capabilities of his model can be summarized as follows :

- Communication between interacting processes can be specified only by communication links which are used for message transfer or shared variable access.
- The SMB model does not have any precise timing specification.
- Within a process, the description is much like a high-level language and is not appropriate to describe various concurrency and sequencing of events in a process.

Because of the nondeterminism and the reasons above, symbolic execution can not prove correctness of the implementation and/or the description of most concurrent hardware systems. However, this thesis is worth further study.

10. Software Verification

In [Taylor 78], Fosdick's (see bibliography) "live variables and available set" concept is slightly modified and, together with "process augmented flow- graph"(PAF), is used to specify and test correct data-flow between concurrent processes.

Their approach is a static analysis of concurrent processes. They define global variables as variables which are accessed by more than one process. First, they construct PAFs for every process, and connect them in such a way that every time a process tries to access a global variable, the global variable must be "available".

As the author mentioned, sometimes it is impossible to constructed a process augmented flow-graph if we include complex process-synchronization primitives which might affect the status of other processes.

This method is valid only for high-level language programs which are not affected by the external (to the program environment) world. That is, all the variables referenced by the processes must be produced by one of the concurrent processes and be available prior to the references. This model is not powerful enough to model concurrent processes.

[Francez 78] classifies variables according to their usage for communication with the external world. The halting problem, dead-lock detection and process synchronization correctness proofs are done using the predicate-based event model of concurrent programs.

This model is very similar to McFarland's hardware behavioral model except that this model fits concurrent software programs better.

11. Software Test Data Selection

[Howden 75] describes how to decompose a program into classes of paths, and then how to test one path in each class. He does this by constructing a "description tree" similar to a control flow graph. This does not seem to be directly applicable to the problem of hardware error detection because he does not seem to be able to handle arbitrary "GO-TO's" and the notion of program termination is important for construction of the description tree.

In [Howden 78], a complete set of functional tests are to be constructed for each of the functions which are part of a program's design. Functions correspond to relatively independent pieces of code. Functional testing is more difficult than structured testing.

[Goodenough 75] proposes to use testing instead of formal proof as means of demonstrating the absence of errors in a program. The authors outline a methodology for the selection of test data inputs since exhaustive testing is usually impossible. They then talk about criteria for selecting such data as follows. Let D be the input domain of the program F,T => D the test set, the predicate Complete(T,C) means that T satisfies criterion C (C can also be a set of criteria). Two main criteria are formally defined for this purpose: reliability and validity. Then the authors come up with the fundamental theorem of testability which states that if a test set is valid and reliable and its execution shows no errors, then the tested program contains no errors. The authors then present an example of program errors in a text formatter. Then a design methodology is outlined which consists of writing a table of what they call test predicates, which are conditions that are relevant to the correct operation of the program. The table should contain all of the possible combinations of the inputs for each of the predicates as a first step, then some thought should be applied to modify the table and make the testing easier. Next, a theory of testing is presented which suggests the partitioning of the input domain into related class sets E(C1) where C1 is a subset of C, the set of predicates. Based on this concept, the definitions of validity and reliability of C are given.

The theory presented in this paper can be applied towards hardware testing. The concepts of completeness, validity and reliability are general and extend beyond the field of software. It might also be possible to find a way of using the McFarland model and its predicates for the construction of the test sets. The theory is still not completely formalized and some work need to be done to get a solid basis for a general testing methodology.

[Howden 82] starts with a description of different approaches to the generation of test data:

- Functional testing treats the program as a black box function and tests the code against the specifications for the function.
- Structural testing requires tests to be constructed that results in the execution of components of the program (each branch must be tested).
- Error-based testing involves the construction of tests which are designed to uncover specific errors or classes of errors.
- Mutation testing requires the definition of a set of mutation transformations which, when applied to components of a program, introduce errors of certain types into the program. (This is the core of the first part of the paper).

The author differentiates between the previous definition of mutation testing, which he calls strong mutation and the testing based on the mutations of single components of a program which he calls weak mutation testing. The author defines 5 types of mutations: variable reference, variable assignment, arithmetic expression, arithmetic relation and Boolean expressions. These types are not mutually exclusive. In the second part of the paper, the author talks about the effectiveness and completeness of test sets. Effectiveness, as he defines it, is closely related to the reliability concept presented in [Goodenough 75]. The completeness concept is also similar to the one discussed in the mentioned paper.

12. Software Correctness

In [Owicki 76], P.Brinch Hansen's concurrent PASCAL is used to model concurrent processes. Hoare's deductive system description technique is used to verify correct sequencing (parallelism, mutual exclusion and dead-lock detection) of the concurrent processes.

This method is not as helpful for hardware modeling since this method can only be applied to structured processes. Hoare's deductive system is very useful to prove the logical correctness of sequential program behavior. However, though the method suggested here is a superset of Hoare's deductive system, it is not capable of describing unstructured concurrent processes. For example, assume that we have "concurrent FORTRAN" with all the capabilities of FORTRAN and concurrency control capabilities of "concurrent PASCAL". Then, this method is not valid for proving correctness of concurrent processes described in such a language. In fact, most concurrent hardware processes and software processes with preemptive scheduling have unstructured control flow.

In [Feldman 79], each distributed process is modeled as a module (a procedure). Modules communicate through public slots (global variables used to pass parameters between procedures). He introduces a "Pending" construct (very similar to the "Wait" construct of concurrent PASCAL).[1] Assert and Cause constructs are used as process control primitives. These primitives are very similar to the Await and Signal constructs of concurrent PASCAL and are useful for process synchronization. Some primitives such as Cause and Assert are useful for description of process synchronization.

[Geller 78] proves that you can generalize from specific test data if you have "covered" the "function" that you are testing with certain test data. He uses a

[1]This feature, with a "receive" statement, can be replaced with one ACCEPT construct of ADA.

combination of predicates to cover the function then tests each predicate separately.

[Shaw 78] presents a notation called "flow expressions" based on regular expressions. There is a cyclic operator which indicates an infinite loop, and an operator called "closure" can be used to indicate dynamic initiation and termination of processes. However, all process sequences seem to be identical in this model. Synchronization is achieved with locks and signals. Membership tests can be formulated using flow expressions and deadlock can be detected. In general, the problem of equivalence between two flow expressions is probably undecidable with the exception of closure, waits and signals. The closure operator can probably be used/extended for Concurrent Behavior Expressions.

13. Software Errors

[Glass 81] states that most errors occur because the program logic is not complex enough. Reuse of variables for different purposes may cause a failure to reset a variable to its original value, resulting in an error. Errors spawned by correction of others are also a problem. There are some other categories of errors, corresponding loosely in several cases to our classification of hardware errors. We estimate that 14% of errors reported here are directly attributable to rule violations, and another 5% to inadequate requirements specifications. A third category, omitted logic, seems to be a combination of rule violation and inadequate requirements specifications, and forms about 30% of reported errors. All but one of the error classes are what we reported as "logical errors". One class, occurring 3% of the time, was reported as a "concurrent error".

[Rubey 75] describes software errors and their discovery. Errors are classified according to the type of computer operation with which they are associated and when they are discovered. The types of errors reported on is a useful classification but may not have the same distribution with respect to hardware errors. However, the error types correspond roughly to our classification of logical errors. They make several pertinent statements: " ... there is no single reason for unreliable software and no single validation tool or technique is likely to detect all types of errors." They claim that program analysis detects errors earlier than execution does.

References

[Brand 83] Brand, D. and Zafiropulo, P.
On communicating finite-state machines.
JACM 30(2), April, 1983.

[Cory 81] Cory, W. E.
Symbolic Simulation for Functional Verification with ADLIB and SDL.
In *Proceedings of the 18th Design Automation Conference*, pages 82-89. ACM SIGDA and IEEE Computer Society DATC, June, 1981.

[Feldman 79] Feldman, J. A.
High Level Programming for Distributed Computing.
CACM 22(6), June, 1979.

[Foulk 80a] Foulk, P. W. and O'Callaghan, J.
Formal description of computational structure in AIDs.
In *IEE Proceeding Vol.127, No.2.* IEE, March, 1980.

[Foulk 80b] Foulk, P. W. and O'Callaghan, J.
AIDs - an integrated design system for digital hardware.
In *IEE Proceeding Vol.127, No.2.* IEE, March, 1980.

[Francez 78] Francez, N. and Pnueli, A.
A proof method for cyclic programs.
Acta Informatica 9:133-157, 1978.

[Geller 78] Geller, M.
Test data as an aid in proving program correctness.
Communications of the ACM 21(5), May, 1978.

[Glass 81] Glass, R.
Persistent software errors.
In *IEEE trans. on Software Engineerring.* IEEE, March, 1981.

[Goodenough 75] Goodenough, J. B. and Gerhart, S. L.
Toward a Theory of Test Data Selection.
IEEE Transactions on Software Engineering SE-1(2), June, 1975.

[Hailpern 80] Hailpern, B. T. and Owicki, S. S.
Verifying Network Protocols Using Temporal Logic.
Technical Report, Computer systems lab., Stanford University, June, 1980.

[Hitchcock 82] Hitchcock, R. B., et. al.
Timing Analysis of Computer Hardware.
IBM Journal of Research and Development 26(1):100-105, Jan., 1982.

[Howden 75] Howden, W. E.
Methodology for the Generation of Program Test Data.
IEEE Transactions on Computers C-4(5), May, 1975.

[Howden 78] Howden, W. E.
Functional program testing.
In *Proceedings of COMPSAC 78 Computer Software and Applications Conference*, pages 321-325. IEEE, November, 1978.

[Howden 82] Howden, W. E.
Weak mutation testing and completeness of test sets.
IEEE Transactions on Software Engineering SE-8(4), July, 1982.

[Huebner 79] Huebner, D.
System Validation Through Automated Requirements Verification.
In *Proceedings of COMPSAC the IEEE Computer Society's Third International Computer Software and applications Conference*, pages 222-227. IEEE, November, 1979.

[Lauer 80] Lauer, P. E.
Project on the Design and Analysis of Highly Parallel Distributed Systems.
Technical Report, the University of Newcastle upon Tyne, December, 1980.

[Leinwand 82] Leinwand, S.
Logical Correctness by Construction.
In *Proceedings of the 19th Design Automation Conference*, pages 825-831. ACM SIGDA and IEEE Computer Society DATC, June, 1982.

[McFarland 83] McFarland, M. and Parker, A.
An Abstract Model of Behavior for Hardware Description.
IEEE Transactions on Computers C-32(7):621-637, July, 1983.

[Moszkowski 82] Moszkowski, B.
A temporal logic for multi-level reasoning about hardware.
Technical Report STAN-CS-82-952, Department of Computer Science, Stanford University, December, 1982.

[Owicki 76] Owicki, S. and Gries, D.
An Axiomatic Proof Technique for Parallel Programs.
Acta Informatica 6:319-340, 1976.

[Pitchumani 81] Pitchumani, V.
Methods of Verification of Digital Logic.
PhD thesis, Syracuse University, 1981.

[Pitchumani 82] Pitchumani, V. and Stabler, E.P.
A Formal Method for Computer Design Verification.
In *Proceedings of the 19th Design Automation Conference*, pages 809-814. ACM SIGDA and IEEE Computer Society DATC, June, 1982.

[Rault ??] Rault, J., Avenier, J., Michard, J. and Mutel, J.
Verification of LSI Digital Circuit Design.

[Rubey 75] Rubey, R. J., Dana, J. A. and Biche, P. W.
Quantitative aspects of software validation.
IEEE Transactions on Software Engineering SE-1(2):150-155, June, 1975.

[Shaw 78] Shaw, A.
Software description with flow expression.
IEEE Transactions on Software Engineering SE-4(3):242-254, May, 1978.

[Taylor 78] Taylor, R. N. and Osterweil L. J.
A facility for verification, testing and documentation of concurrent process software.
IEEE , 1978.

[Tran 82] Tran, A., Forsberg, R. and Lee, J.
A VLSI design verification strategy.
Technical Report 4, IBM, July, 1982.

[Umrigar 83] Umrigar, Z. D. and Pitchumani V.
Formal verification of a real-time hardware design.
In *Proceedings of the 20th Design Automation Conference*, pages 221-227. IEEE, 1983.

[VanMierop 79] VanMierop, D.
Design and Verification of Distributed Interacting Processes.
PhD thesis, Computer Science Department, University of California, Los Angeles, March, 1979.

[VanOost 81] E.M.J.C. VanOost.
Multi-processor system description and simulation using structured multi-programming languages.
Computer Architecture News, ACM 9(2), April, 1981.

Appendix 2

Documentation and Program Listing for V Collision Detector

By D. Knapp

31 October 1984

Table of Contents

1. The Disjoint-Path Algorithm

The disjoint-path algorithm runs as follows:

1. If R_1 and R_2 have a common source node, check the type of that node. If it is an and-fork they are in conflict; otherwise not.

2. If R_1 and R_2 do not have a single common source node, then the set of disjoint execution paths must be checked:

   a. Set the lists {C}, {U}, {W}, and {V} to NIL.

   b. Mark the source node of R_1 with the tag 'a' and the source node of R_2 with a 'b'. This is the initial set of 'current nodes'.

   c. If a current node is a global source node, union its tag into {V}. If this set ever has cardinality greater than one, stop: R_1 and R_2 are in conflict, because the two source nodes may become active simultaneously.

   d. If a current node is a simple node, put it at the end of {C} and go to step c.

   e. If a current node is a fork node, put it at the end of {U} and go to step c.

   f. If a current node is a join node, sort it into {W} in reverse order, the order being defined during the original formation of predecessor sets (above), and go to step c.

   g. If there is no current node, create a new one.

In order to create a new current node, the following procedure is used:

1. If the combined cardinality $|C| + |U| + |W| + |V|$ is one, stop. There is only one path of execution if this is the case, and R_1 and R_2 are not in conflict.

2. Otherwise, if {C} is nonempty, then

   a. Remove the first element of {C}. Call it D.

   b. Find the in-arc A of D. Mark it with the same tag D had.

3. Otherwise, if {U} is nonempty, then

   a. Examine the first element of {U}. Call it D.

b. Find an unmarked in-arc A of D. Mark it with the same tag D had.

c. If all in-arcs of D are now marked, remove D from {U}.

4. Otherwise, if {W} is nonempty, then

a. Remove the first element of {W}. Call it D.

b. Find the in-arc A of D. Mark it with the same tag D had.

5. Check the tags of the out-arcs of the source node S of A. If the tags form an incompatible set, then stop; R_1 and R_2 are in conflict. If not, then S is the new current node. Mark it according to the following rule:

a. If S is an or-fork node, and at least one of its out-arcs is marked with an 'a' and at least one out-arc with a 'b', then mark S with an 'ab', signifying that it can result in either R_1's or R_2's execution, but (because it is an or-fork) not both. If any out-arc is an 'ab', mark S with an 'ab' regardless of the other arcs. If all marked out-arcs have the same tag, use that tag. Unmarked out-arcs do not count, because the imposed ordering makes it impossible for any unmarked out-arc to be a predecessor of either R_1 or R_2.

b. If S is an and-fork and it has one or more 'b' out-arcs and one or more 'a' out-arcs, or an 'ab' out-arc and any other marked out-arc, stop. R_1 and R_2 are in conflict, because it is possible for both to be executed at once. If all marked out-arcs of S are either 'a' or 'b', then the mark of S is the same as that of the out-arcs.

c. If S is a simple node or a join node, then the mark of S is the same as the mark of the out-arc of S. Because no node can be both a fork and a join simultaneously, there is only one out-arc.

2. Correctness and Complexity

This section addresses the arguments that can be made about the correctness and complexity of the algorithm.

The correctness of this approach hinges on the backtracing-and-marking algorithm, which is also the most expensive part of V.

First, we will consider the number of (R_1,R_2) pairs. The maximum cardinality of this

set (i.e. |C1|) is $|R|^2$. Since in constructing C2 we consider each (R_1,R_2) element of {C1} only once, the construction of C2 will terminate in at most $O(R^2)$ such considerations. Let us then turn to the question of the cost and termination of checking a single (R_1,R_2) pair.

Because the backtracing is done in a DAG, we can argue that the algorithm will terminate, given that each arc will be considered only a finite number of times. In fact, each arc is considered only once, when it is transited. It cannot be considered again, because its sink node will be removed from {C}, {U}, or {W}, and once removed it cannot be replaced in {C}, {U}, or {W}. The complexity of checking an (R_1,R_2) pair is therefore at worst the number of arcs times the complexity of transiting an arc.

The reason a node cannot be put back into one of those lists once it has been removed from any of them is that the reversal of the DAG ordering is preserved when elements are placed on the {C}, {U}, and {W} lists. That preservation, in fact, is the reason that current nodes are split into these three lists. To visit a node twice would require that this ordering be violated.

The elements of {C}, {U}, and {W} form a 'wavefront'. We will now argue that (1) the wavefront begins life in an ordered state, and that (2) all of the operations that change the membership of the wavefront preserve the ordered state.

The wavefront begins in an ordered state because it consists of either one or two elements, the source nodes of R_1 and R_2. If they are one and the same node, the singleton is 'ordered' in the sense that there is no other node 'after' it that should be 'before' it; if there are two nodes, then they will either fall into the same list (i.e. they are of the same class) or they will not. If they do not fall into the same list, each is the only element of its list; there is no problem there. If they fall into the same list, then they are placed in that list in order, by referral to the sequence numbers given them in the initial (forward) transit of the DAG. It can be shown, by quite similar arguments, that those sequence numbers preserve the DAG's partial ordering. If the construction of {C1} is done properly, the pairs (R_1,R_2) will be in order, so it is merely necessary to

place the source of R_1 after that of R_2.

The wavefront stays ordered because of the way in which nodes are chosen for transition. Recall that a member of {W} will only be chosen if both {U} and {C} are empty; this is why.

If the head of {W}, called P, is chosen, then both {U} and {C} are empty. transition of the sole in-arc of P cannot reach a higher-numbered node than P; therefore it cannot violate the ordering if it is sorted into {W} behind P. It may already be in {W}; this is not a problem, as nothing further needs to be done. If it is placed in {C} or {U} it will be the only element of those lists and therefore cannot violate the ordering. Because {U} and {C} are empty, and all other elements of {W} precede P, any unmarked out-arcs of P can never be reached. Because they can never be reached, P will never be seen again once it is removed. Because it will never be seen again, it cannot be placed in any of the lists in violation of the ordering.

If the head of {U}, again called P, is chosen, it must be because {C} is empty. {U}, it will be recalled, is composed of join-nodes; P therefore has more than one in-arc. An in-arc A is chosen at random from the set of P's unmarked in-arcs. A is then marked and transited. If P has no more unmarked in-arcs, it is removed from the list. If P's source node S falls into {C} it will be the only element of {C}. If S falls into {W}, it will be sorted into {W}, so order is preserved.

P's source node S, if it falls into {U}, will fall at the tail of {U}. Will putting S at the tail of {U} necessarily preserve the ordering of {U}?

1. Either S and P are the only elements of {U}, in which case order is guaranteed, or there are intermediate elements {I}. Any element of {I} must have been reached and marked in order for it to have been placed in {U}.

2. The only way in which an element of {I} can have been reached and still precede S is if there is a member of {W}, called O, between it and S, through which a mark was transmitted in a previous step.

3. But the only way in which O can have been transited out of would have been if both {U} and {C} were empty at the time; therefore there must be a path

from P to some member of {W} called Q. Otherwise P cannot have been transited at all.

4. By the fact that Q was reachable from P, and P from O, Q must be strictly 'later' than O; therefore because {W} is in reverse order, Q would have preceded O in {W}, and there is a path of {C} and {U} elements from P to Q. O therefore cannot have been transited by the time S is put into {U}; therefore such an element of {I} cannot exist.

If the head of {C}, called P, is chosen, it is removed from {C} and its (single) in-arc marked and transited. The node S so reached is then either appended to {C} or {U}, or sorted into {W}.

If S is a fork node, it is sorted into {W}, which preserves the ordering. If it is a join node, it is appended to {U}, under an argument similar to the argument given above.

If S is a simple node, then it is appended to {C}. Because a member of {C} will always be transited if it exists, chains of simple nodes will be transited until either a fork, a join, or a global source node is reached; no other types of nodes will be transited until this happens. Furthermore, {C} can never consist of more than two simple nodes, and those nodes are necessarily unordered with respect to one another.

The reason C can consist of at most two unordered nodes is that the sources of both R_1 and R_2 can be simple. In that case, both are placed in {C}. But they must be unordered, because that would imply that the predecessor's out-arc was a predecessor of the successor; i.e. that the pair (R_1, R_2) is not a member of {C1} and will not be considered at all.

Now consider the fate of the first node P in {C}. It is removed from {C}. Its predecessor S is either a simple node or it is not. If it is not, it will be placed in either {U} or {W}; if it is, it will be appended to {C}. S cannot be a predecessor (or vice versa) of the now first element of {C} because that would imply that P also had been; but P and the new first element were guaranteed to be unordered. The two elements of {C} will therefore be unordered at least until some other node type is encountered; but when that happens, the new node S is not placed in {C} and {C} has only one element

thereafter.

Because chains of simple nodes are transited before any other nodes are considered, we can guarantee that when another node is considered, {C} is empty and no ordering conflict can arise in {C}. Furthermore, because at most one element can be added to {C} in any transition, and that element will always be deleted in the following transition, {C}'s cardinality can never grow from one to two.

This in turn implies that a chain of simple nodes can be regarded as a single arc for the purposes of ordering {W} and {U} points. Hence transiting a simple node is subject to all of the arguments given above about the orderings of elements of {W} and {U}, with such chains being treated as single arcs.

2.1. Overall Complexity

The time complexity of V is the sum of the time complexities of its serial parts. These are:

1. Forward traversal: O(R), where R is the number of ranges.

2. Predecessor collection: O(R) for each range, when done in conjunction with forward traversal. Complexity for both is therefore $O(R^2)$.

3. Successor formation: $O(R^2)$ (copy operation)

4. {C1} formation: $O(R^2)$.

5. {C2} formation: There are $O(R^2)$ possible elements of {C2}. Backtracing costs O(R) times the cost of transiting a range, which is dominated by the cost of maintaining {W}. By using a clever trick, a vector of all possible elements of {W}, marked by their presence in {W} or absence from it, this can be kept to O(R) for the entire backtrace of one (R_1,R_2) pair. {C2} formation under this method is then $O(R^3)$, which dominates V unless the number of bindings is large.

6. {C3} formation: lookup in {C2} costs unit time if {C2} is properly organized. This must be done for every pair of bindings where the hardware resource is the same; i.e. $O(B^2)$, where B is the cardinality of bindings for a single hardware resource.

3. The V Collision Detector: How To Use It

3.1. General

V is a franz-lisp based program that detects collisions between nmr-bindings and vcr-bindings. It does this by extracting a set of conflicting ranges and then comparing each binding pair to the set of conflicting ranges. It extracts conflicting ranges by the following procedure:

1. Transit the timing/control graph in a forward direction:
    a. Collect the predecessors of each range
    b. 'Crush' chains of simple points into 'superranges'
2. Construct from the predecessor lists the set C1, which is the set of range pairs that are unordered
3. For each pair in C1, determine if the pair is really in conflict:
    a. Assign the first the tag 'a', the second the tag 'b'.
    b. Propagate tags backwards along all possible paths until either
        i. An and-fork with both a and b (or ab and anything; see below) out-arcs is found, or
        ii. All of the paths have merged at a single point and condition (1) hasn't been met.

    (an 'ab' out-arc is generated when an 'a' and a 'b' meet at an or-fork; the tag means 'either range a or range b can descend from this arc, but not both simultaneously'. If an ab arc meets any other marked arc at an and-fork, there is a potential collision.)

V will automatically write out the file 'CollisionList' when it terminates if there are any collisions. Otherwise it will allow the user to examine the sets of objects, but will not write anything out unless told to do so.

3.2. Running V

It is currently set up as a stand-alone program. It is run by the following sequence, shown with example commands slightly indented.

1. Invoke lisp.

```
cse> lisp
```

2. Load the V source.

```
--> (load '/usr/adam/v/v)
```

3. V will ask you for the name of an input file.

```
What input file?
/usr/adam/v/lib/examples/tc1
```

4. Some messages will appear on the screen as V executes, so you will know it isn't dead.

```
Reading in the data file...

Crushing the graph....
```

5. The last message V will write to you is a notification that collisions exist (if they do). Otherwise, it will simply return to the lisp top level.

```
There are collisions listed in CollisionList

-->
```

6. The file CollisionList will automatically be written if there are any collisions. The lisp interpreter can either be exited at this point by the command control-D, or the top leVel can be used to examine other data structures. It is NOT RECOMMENDED that the V program be run again without reloading the interpreter; V leaves garbage lying around and the results are unpredictable.

```
--> ^D

Goodbye

cse>
```

3.3. Ancillary Information

Aside from the file CollisionList, other files can be written containing other information.

If you want any of the following lists:

1. **C1**, the list of range pairs that are not ordered (i.e. that are in 'parallel' with one another), but which may exclude one another

2. **C2**, the list of range pairs that actually are in conflict a subset of C1)

3. **C3**, the list of colliding bindings

4. **ralist**, the list of ranges

5. **polist**, the list of points

6. **molist**, the list of modules (or carriers; V does not distinguish between them)

7. **cralis**, the list of superranges

You can print out any or all of these by using the correct lisp commands, e.g.

```
--> (pp (F foo.baz) C2)
```

which will write the list C2 into the file 'foo.baz' in pretty-printed form (This is highly recommended. Pretty-printing makes the files human-readable; otherwise it's pretty difficult going.).

It is also possible to examine the structure of the timing graph and other interesting things by examining the property lists of the objects in the lists. A function called 'rbk' is provided for this purpose:

```
--> (rbk molist 1)
```

This will write out the property lists of all of the modules (or carriers, as the case may be), to a file called 'f1'. The numeric parameter can range from 1 to 3; the output files will be called f1, f2, and f3 in that case. Because C1, C2 and C3 are lists of lists, and rbk expects to see a list of objects, it will fail if you give it one of these names as a parameter. The way to look at the property lists of the objects in those lists would be either

1. Use elements of C1, C2 or C3 as parameters of rbk:

   ```
   --> (rbk (car C2) 1)
   ```

   which will print out the property lists of the two ranges (range1, range2) of the first pair of conflicting ranges.

2. Write out the property lists of the elements and also the contents of the list of lists:

```
--> (rbk molist 1)
.
t

--> (pp (F C3lis) C3)
```

which will write out the property lists of the modules into the file 'f1', and the list C3 into the file 'C3lis'.

Because V wants to see unique object names, and because the user cannot be expected to know all of the object names that are already in V, V assigns rather unusual names to objects when it reads in the data. These names are what you will see when you write out the files; at some later stage, the names will be restored to their original state. For the moment, it is still possible to recognize the names you put into V, by knowing how it constructs the names you get out. A few examples will demonstrate the transformation of

1. Taking the first two letters of the object type,

2. Append to that the string '^$',

3. Append to that the original object name.

Hence the node named 'Foobaz' will be converted to 'no^$Foobaz', and the range 'barz' will be converted to 'ra^$barz'.

In the lists of cralis, C1 and C2, an additional transformation is applied because these are lists of superranges or superrange pairs. This transformation is:

1. Take the string 'sr^$'

2. Append to that the (internal) name of the first range in the superrange.

Hence, if there was a range called 'Fubar' in the input data file, it would first be converted to 'ra^$Fubar'. If, furthermore, it was the first range in a superrange, the superrange would be named 'sr^$ra^$Fubar'.

The property lists of objects contain the following fields. These property lists are the first elements taken from an actual test run of V; this is what it looks like, with the

exception of a little reformatting done to make it fit the tabular format used here. A 'private variable' is one which is internal to V; its state is useful for diagnostic purposes, but of little concern to the user.

cralis element:

property name	value	meaning
tsuc	nil	private variable. usually nil.
succ	nil	private variable. usually nil.
pred	(sr^$ra^$gh sr^$ra^$cd sr^$ra^$ce sr^$ra^$no sr^$ra^$jk sr^$ra^$jl sr^$ra^$bc sr^$ra^$bi sr^$ra^$ab)	set of all superranges that precede this range
rsuc	po^$q	point that succeeds this range
eqset	(ra^$pq)	ordinary ranges represented by this superrange (usually has more than one element)
rprc	po^$p	point that precedes this range
sqno	10	private variable. order of traversal.
name	sr^$ra^$pq	the name of the superrange

molist element

property name	value	meaning
brex	((no^$n4 mo^$A ra^$hp) (no^$n3 mo^$A ra^$fg) (no^$n2 mo^$A ra^$dg) (no^$n1 mo^$A ra^$bc) (no^$n0 mo^$A ra^$ab))	list of all bindings of the module or carrier
name	mo^$A	name of the module or carrier

ralist element

property name	value	meaning
LE	10	latest possible ending time
EE	6	earliest possible ending time
LB	0	latest possible beginning
EB	0	earliest possible beginning
	(The four quantities above may not be present. If this happens it is because either (1) somewhere in the graph a range does not have ub/lb properties, or (2) the graph has more than one global source point. In both of these cases, these times will be invalid.)	
eqset	sr^$ra^$ab	the superrange that represents this range
sqno	1	private. order of traversal.
ub	10	upper bound on the length of this range
lb	6	lower bound on L. of this range
rsuc	(po^$b)	successor point
rprc	(po^$a)	predecessor point
name	ra^$ab	name of the range

polist element

property name	value	meaning
wno	1	private. fork points only. order of traversal
SucSupRan	(sr^$ra^$bc sr^$ra^$bi)	set of successor superranges

PreSupRan	(sr^$ra^$ab)	predecessor superranges
rsux	(ra^$bi ra^$bc)	successor ranges
rprx	(ra^$ab)	predecessor ranges
ptype	afork	type of the point
name	po^$b	name of the point

3.4. Input Data Format

Only the following types of lines may be inputted to V at this time. A more robust version is in the works; in the meantime the use of emacs editor macros and Agis data files is highly recommended. Examples of each type are given between the general templates. The examples are slightly indented.

CONVENTIONS:

N is a generic node name
M is a generic module name
R is a generic range name
Strings in lower case are assumed to be quoted, e.g. "foo" <-> 'foo
PTYPE is an element of
    {simple afork ajoin ofork ojoin} *NB* they are quoted!
STYPE is an element of {src sink}
RTYPE is an element of the set of ascii representations of real numbers.
$ is an EOL with other trash in front of it that will be ignored.

Here are the valid lines. They may be repeated indefinitely.

```
mk PTYPE P $

  mk simple a ;
  mk afork b ;
  mk ofork c ;

bind P R STYPE $

  bind a ab src ;
  bind b ab sink ;
  bind b bc src ;

real u R RTYPE $ ;upper bound

real l R RTYPE $ ;lower bound

  real l ab 6 ;
  real u ab 10 ;
  real l bc 6 ;
  real u bc 10 ;
  real l cd 6 ;

xbind N M R $

  xbind n19 E ce ;
  xbind n20 E ef ;
  xbind n21 E fg ;
  xbind n22 E gh ;


xbind V C R $

  xbind n19 G ce ;
  xbind v20 G ef ;
  xbind v21 G fg ;
  xbind v22 G gh ;
```

Example data files are to be found in /usr/adam/lib/v/examples.

FILES:

/usr/adam/v/v.l	lisp source code
/usr/adam/v/examples/tc1	example data files
/usr/adam/v/examples/tc2	
/usr/adam/v/examples/ralist.ex	sample of (rbk ralist .)
/usr/adam/v/examples/C2	sample of (pp (F C2) C2)
/usr/adam/v/examples/C3	sample of (pp (F C3) C3)

WARNINGS:

1. Always reload the lisp interpreter between runs.

2. Does not accept straight Agis data files. Input in general is very fragile.

PROBLEMS: D W Knapp, SAL 349, USC 90089-0781, or
knapp%usc-cse@csnet-relay.arpa

4. Program Listings

```
;this pgm performs the collision detection test.
;it is run just by loading, at the end it invokes the
;fcn R immediately below, which invokes all the other
;fcns.
;It is written in Franz Lisp. There are important differences
;between FL and other lisps; be careful in porting it. For
;one thing, such atoms and function IDs as Foo and foo are
;NOT THE SAME and case folding will get you into trouble in
;this pgm.
;this pgm runs under Berkeley 4.2bsd Unix.
;NB there is also some dead code, e.g. fcns stophere
;and mkdots, which were used in debugging the pgm. They were
;so useful I left them in.
;
;documentation is to be found in /usr/adam/doc/v.doc
;
;problems to knapp%usc-cse@csnet
-relay.arpa
;
;signed, D W Knapp (dwk) 9-84


(defun R ()
;this is the uppermost fcn of the entire pgm, it does everything.
;be careful, there are lots of side effects.
  (echo "Off and Running!")
  ;initialize the oblist data structures
  (setq polist nil molist nil)
  (setq S nil ralist nil bakwrdralis nil)
  (sequence 0)
  (echo "what data file?")
  (setq tfil (read))
  (echo "reading it")
  (read-data-struc tfil)
  (echo "crushing the graph")
  (CG)
  (echo "taking the forward TC")
  (TC2)
  (echo "building C1")
  (mkC1)
  (echo "building C2")
  (mkC2)
  (echo "checking for collisions")
  (ckbn)
  (cond (collbool
     (echo "there is at least one collision listed in CollisionList")
     (pp (F CollisionList) C3)))
]
```

```
(defun read-data-struc (filename)
;r-d-s reads in design data structures
;this version ignores everything but timing
;and to a certain extent structure
;a more extensive version is found in ~/src/x/x.l
    (prog ()
      (setq p1 (infile filename))
      (define-loop)]

(defun define-loop ()
;read in an agis command and put it into the data structure,
;then loop back until EOF
  (prog (cmd)
   loop (setq cmd (read p1 'EOF))
      (cond
       ((eq 'EOF cmd) (return 'done))
       ((null cmd) (go loop))
       ((eq cmd 'mk) (makit))
       ((eq cmd 'bind) (bindit))
       ((eq cmd 'real) (realit))
       ((eq cmd 'xbind) (interbind))
       (t  (zapline)))
      (mkdots 10)
      (go loop)]

(defun makit ()
;create a new object
    (enlist (read p1 'EOF) (read p1 'EOF))
    (zapline)
]

(defun bindit ()
;create a binding between objects
  (prog (tmpponm tmpranm tmppin)
    (setq tmpponm (dat 'po (read p1 'EOF)));get the names
    (setq tmpranm (dat 'ra (read p1 'EOF)))
    (setq tmppin (read p1 'EOF));src or sink
    (set tmpponm nil);be sure they are on the oblist
    (set tmpranm nil)
    [cond
      ((eq tmppin 'sink)
         (putprop tmpponm (cons tmpranm (get tmpponm 'rprx)) 'rprx)
         (putprop tmpranm (cons tmpponm (get tmpranm 'rsuc)) 'rsuc))

      (t (putprop tmpponm (cons tmpranm (get tmpponm 'rsux)) 'rsux)
         (putprop tmpranm (cons tmpponm (get tmpranm 'rprc)) 'rprc))]
    (zapline)
]

(defun interbind ()
;create an interview binding
  (prog (tmpnonam tmpmonam tmpranam L)
    (setq tmpnonam (dat 'no (read p1 'EOF)))
```

```
    (setq tmpmonam (dat 'mo (read p1 'EOF)))
    (setq tmpranam (dat 'ra (read p1 'EOF)))
    (setq L (list tmpnonam tmpmonam tmpranam))
    (putprop tmpmonam (cons L (get tmpmonam 'brex)) 'brex)
    (zapline)
]

(defun enlist (ptyp pnom)
;enlist puts the pnom on the oblist with a rather wierd name
;to avoid accidental collisions,
;and attaches the ptyp as a property of it
  (prog (tempvar)
    (setq tempvar (dat 'po pnom))
    (set tempvar nil)
    (putprop tempvar ptyp 'ptype)

(defun realit ()
;reads in a real-valued parameter of an object
  (prog (tempvar n1 flagg)
    (setq flagg (read p1 'EOF))
    (setq n1 (read p1 'EOF))
    (setq tempvar (dat 'ra n1))
    (set tempvar nil)
    (cond
      ((eq flagg 'u) (putprop tempvar (read p1 'EOF) 'ub))
      ((eq flagg 'l) (putprop tempvar (read p1 'EOF) 'lb))
      (t (echo "what the hell?")))
    (zapline)
]

(defun dat (type name)
;creates an odd object name and adds the object to a list of similar objs
  (prog (tmpvar)
    (setq tmpvar (concat type '^$ name))
    (cond ((eq type 'po) (setq polist (insert tmpvar polist nil t))))
    (cond ((eq type 'ra) (setq ralist (insert tmpvar ralist nil t))))
    (cond ((eq type 'mo) (setq molist (insert tmpvar molist nil t))))
    (putprop tmpvar tmpvar 'name)
    (return tmpvar)
]

(defun rbk (listname pntr)
;run back the contents and plists of a list
    (setq t4  (mapcar 'tell-about listname))
  (cond
    ((eq pntr 1)
      (pp (F f1) t4))
    ((eq pntr 2)
      (pp (F f2) t4))
    ((eq pntr 3)
      (pp (F f3) t4))
    ((eq pntr 4)
      (pp (F f4) t4))
```

```
    (t (pp t4))
]

(defun tell-about (name)
  (prog (temp)
    (setq temp (plist name))
    (return temp)
]


(defun errare macro (L)
;say something to the user
   (list 'prog 'nil (cons 'patom (cdr L)) '(terpri) '(return nil)]

(defun echo (echoatom)
;send an atom to the user
  (terpri)
  (cond (echoatom (errare echoatom))]

(defun allmarkedp (pointname fieldname flagg)
;returns true if all of the flagg arcs are marked
  (prog (arx)
    (setq arx (get pointname flagg))
   Loop
    (cond
      [(null arx) (return t)]
      [(get (car arx) fieldname)
          (setq arx (cdr arx)) (go Loop)]
      [t (return nil)]
]

(defun allmarkedq (pointname fieldname flagg markno)
;a little more specific, it examines a mark's identity as well
  (prog (arx)
    (setq arx (get pointname flagg))
   Loop
    (cond
      [(null arx) (return t)]
      [(eq (get (car arx) fieldname) markno)
          (setq arx (cdr arx)) (go Loop)]
      [t (return nil)]
]

(defun forkp (point)
;is a node a fork node
  (or (eq (get point 'ptype) 'afork) (eq (get point 'ptype) 'ofork)))

(defun joinp (point)
;is it a join
  (or (eq (get point 'ptype) 'ajoin) (eq (get point 'ptype) 'ojoin)))

(defun simplep (point)
;is it a simpel pt
```

```
  (eq (get point 'ptype) 'simple))

(defun stophere ()
;stop and enter a read-eval-print loop
;useful for debugging
  (prog (useritem)
  loop (terpri) (echo '>>') (setq useritem (read))
       (cond ((eq useritem 'go) (return nil))
             (t (print (eval useritem)) (go loop))
]
(defun mkdots (multiplier)
;useful for debugging
 (cond
  ((greaterp SS$B multiplier)
    (setq SS$B 0)
    (patom '.')
    (drain)
    (setq SS$A (+ 1 SS$A))
    (cond ((greaterp SS$A 70) (setq SS$A 0) (terpri))))
  (t   (setq SS$B (+ 1 SS$B)))
]


(defun CG ()
;  this function crushes a graph by removing all simple points.
;  DATA structures:
;    operates on ralist and polist
;    creates cralis
;      elements of cralis have:
;       eqset: list of ralist elements
;       sqno: eq. of seqnumber
;       prp:pred. point (polist element)
;       sup:suc point '  '
;       cnf:set of cralis conflict ranges
;       prr (, suc):set of predecessor (successor) cralis elements

  (prog ()
     (setq cralis nil)
     (setupUs)
   Loop
     (mkdots 1)
     (cond
       [(car C)
          (setq p (car C)) (setq C (cdr C))
          (hopc p)
          (go Loop)
       ]
       [(car U)
          (setq p (car U))
          (hopc p)
          (cond ((allmarkedp p 'sqno 'rsux)
              (putprop p (sequence 5) 'wno)
              (setq U (cdr U))))
```

```
          (go Loop)
        ]
]
(defun setupUs ()
;set up the U list
  (prog (P latcnt)
    (setq P polist)
    (setq C nil U nil)
    (setq latcnt 0)
   Loop
    [cond ((null P)
        (setq lattice$bool (not (greaterp latcnt 1))) (return 'done))
      (t
        [cond
          ((null (get (car P) 'rprx))
           (setq latcnt (+ 1 latcnt))
           (cond
             ((eq (get (car P) 'ptype) 'simple)
               (setq C (cons (car P) C)))
             ((forkp (car P))
               (setq U (cons (car P) U)))
             (t (echo "funny source point type: death")))))]
        (setq P (cdr P)) (go Loop))]
]
(defun hopc (point)
;this fcn adds in a range or set of ranges to cralis, and (possibly)
;does not.
  (prog (ran aux)
    ;select the range to expand and create the cralis element
    (setq aux (getran1 point))
    ;aux (and thence ran) is an out-arc of point
    (cond ((null aux) (return nil)))
    (setq ran (dat 'sr aux))
    (setq cralis (cons ran cralis))
    (putprop ran (sequence 3) 'sqno)
    (putprop ran point 'rprc)
    (putprop point (cons ran (get point 'SucSupRan)) 'SucSupRan)
    ;add the range to its eqset
  Loop
    (putprop ran (cons aux (get ran 'eqset)) 'eqset)
    (putprop aux (sequence 1) 'sqno)
    (putprop aux ran 'eqset)
    (setabstimes aux)
    ;see if its sink pt is a simple pt
    [cond
      ((simplep (car (get aux 'rsuc)))
          ;if so then add the next range to the eqset and loop
        (setq aux1 (getran1 (car (get aux 'rsuc))))
        (cond
          ((null aux1)
            (putprop ran (car (get aux 'rsuc)) 'rsuc)
            (putprop (car (get aux 'rsuc))
              (cons ran (get (car (get aux 'rsuc)) 'PreSupRan)) 'PreSupRan)
```

```
            (go Lable))
           (t (setq aux aux1)))
          (go Loop))
         (t
           ;otherwise add the pt to the cralis element as a successor
          (putprop ran (car (get aux 'rsuc)) 'rsuc)
          (putprop (car (get aux 'rsuc))
            (cons ran (get (car (get aux 'rsuc)) 'PreSupRan)) 'PreSupRan)
            ;and to either U or C as approp.
          (cond
            ((forkp (car (get aux 'rsuc)))
               (setq U (append U (get aux 'rsuc))))
            ((allmarkedp (car (get aux 'rsuc)) 'sqno 'rprx)
               (setq C (append C (get aux 'rsuc))))]
   Lable
    (putprop ran (merg2 (get ran 'rprc)) 'pred)
]

(defun setabstimes (ran)
;this fcn sets up the ub and lb times of a range, if there is a single
;global source from which it can be derived.
  (prog (pt)
    (cond ((null lattice$bool) (return nil))
      (t
        (setq pt (car (get ran 'rprc)))
        (putprop ran (EBtime ran pt) 'EB)
        (putprop ran (LBtime ran pt) 'LB)
        (putprop ran (EEtime ran pt) 'EE)
        (putprop ran (LEtime ran pt) 'LE)))))

(defun EBtime (ran pt)
;this fcn returns the earliest time the ran can begin
  (prog (preds)
    (setq preds (get pt 'rprx))
    (cond
      ((null preds) (return 0))
      ((eq (get pt 'ptype) 'ajoin)
        (return (UB preds 'EE)))
      ((eq (get pt 'ptype) 'ojoin)
        (return (LB preds 'EE)))
      (t (return (get (car preds) 'EE))))))

(defun LBtime (ran pt)
;this fcn returns the latest time the ran can begin
  (prog (preds)
    (setq preds (get pt 'rprx))
    (cond
      ((null preds) (return 0))
      ((eq (get pt 'ptype) 'ajoin)
        (return (UB preds 'LE)))
      ((eq (get pt 'ptype) 'ojoin)
        (return (LB preds 'LE)))
      (t (return (get (car preds) 'LE))))))
```

```
(defun UB (ranlis flagg)
;this fcn returns the upper bound of the 'flagg fields of the elements
;of ranlis
  (prog (bnd)
    (setq bnd 0)
    Loop (cond
      ((null ranlis) (return bnd))
      ((greaterp (get (car ranlis) flagg) bnd)
        (setq bnd (get (car ranlis) flagg))))
      (setq ranlis (cdr ranlis))
      (go Loop)))

(defun LB (ranlis flagg)
;this fcn returns the lower bound of the 'flagg fields of the elements
;of ranlis
  (prog (bnd)
    (setq bnd (get (car ranlis) flagg))
    (setq ranlis (cdr ranlis))
    Loop (cond
      ((null ranlis) (return bnd))
      ((greaterp bnd (get (car ranlis) flagg))
        (setq bnd (get (car ranlis) flagg))))
      (setq ranlis (cdr ranlis))
      (go Loop)))

(defun LEtime (ran pt)
;this fcn returns the latest time the ran can end
  (prog (ubl)
    (setq ubl (get ran 'ub))
    (cond ((null ubl) (setq lattice$bool nil) (return 0))
      (t  (return (+ ubl (get ran 'LB)))))))

(defun EEtime (ran pt)
;this fcn returns the earliest time the ran can end
  (prog (lbl)
    (setq lbl (get ran 'lb))
    (cond ((null lbl) (setq lattice$bool nil) (return 0))
      (t  (return (+ lbl (get ran 'LB)))))))


(defun sequence (seqnu)
;this fcn generates static sequences for unique counters, marks, etc
  (prog ()
    (cond
      ((eq seqnu 0)
         ;set up the static counters
         (setq SS$1 0 SS$2 0 SS$3 0 SS$4 0 SS$5 0 SS$A 0 SS$B 0)
         (return 0))
      ((eq seqnu 1) (setq SS$1 (+ 1 SS$1)) (return SS$1))
      ((eq seqnu 2) (setq SS$2 (+ 1 SS$2)) (return SS$2))
      ((eq seqnu 3) (setq SS$3 (+ 1 SS$3)) (return SS$3))
      ((eq seqnu 4) (setq SS$4 (+ 1 SS$4)) (return SS$4))
      ((eq seqnu 5) (setq SS$5 (+ 1 SS$5)) (return SS$5))
```

```
      (t (return nil))
]
(defun refersq (seqnu)
;this fcn references static sequences of sequence without changing them
  (prog ()
    (cond
      ((eq seqnu 0) (return nil))
      ((eq seqnu 1) (return SS$1))
      ((eq seqnu 2) (return SS$2))
      ((eq seqnu 3) (return SS$3))
      ((eq seqnu 4) (return SS$4))
      ((eq seqnu 5) (return SS$5))
      (t (return nil))
]
(defun merg2 (point)
;this fcn grabs all of the 'pred fields of the ranges in the 'rprx
;of the point passed, and merges them into a single list
;L is the result list
;ranges is the list of ranges incident on 'point
;preds is the list of predecessors of one of {ranges}
  (prog (L ranges preds)
      (setq L nil ranges (get point 'PreSupRan))
   Loop
      (cond
        [(null ranges) (return L)]
        [t
          (setq L (merg3 (cons (car ranges) (get (car ranges) 'pred)) L))
          (setq ranges (cdr ranges)) (go Loop)]
]

(defun merg3 (arcl1 arcl2)
;this fcn merges two lists in backwards order by sqno
  (prog (L)
      (setq L nil)
    Mloop
      (cond
        [(and (null arcl1) (null arcl2)) (return (reverse L))]
        [(null arcl1) (setq L (append (reverse arcl2) L) arcl2 nil)]
        [(null arcl2) (setq L (append (reverse arcl1) L) arcl1 nil)]
        [(eq (get (car arcl1) 'sqno) (get (car arcl2) 'sqno))
           (setq L (cons  (car arcl1) L))
           (setq arcl1 (cdr arcl1) arcl2 (cdr arcl2))]
        [(greaterp (get (car arcl1) 'sqno) (get (car arcl2) 'sqno))
           (setq L (cons (car arcl1) L) arcl1 (cdr arcl1)]
        [t (setq L (cons (car arcl2) L) arcl2 (cdr arcl2))])
      (go Mloop)
]

(defun TC2 ()
;this fcn takes the tr. closure of cralis.
  (prog (r1 r2 rL crtmp)
;r1 is the range currently under scrutiny
;r2 is a predecessor that is getting r1 put into its 'succ field
```

```
;rL is the 'pred field of r1
;crtmp is a copy of cralis that will get destroyed
    (setq crtmp cralis)
    Loop1
      (mkdots 1)
      (cond
        [(null crtmp) (go Lable)]
        [t (setq r1 (car crtmp))
           (setq rL (get r1 'pred))
           (addsuc r1 rL)
           (setq crtmp (cdr crtmp))
           (go Loop1)])
    Lable ;now reverse the succ lists so they are in reverse-first order
      (setq crtmp cralis)
    Loop2
      (cond
        [(null crtmp) (return 'done)]
        [t (setq r1 (car crtmp))
           (putprop r1 (reverse (get r1 'succ)) 'succ)
           (setq crtmp (cdr crtmp))
           (go Loop2)]
]
(defun addsuc (ran ranlist)
;this fcn cons's ran onto the succ lists of all the ranges in ranlist
  (prog (r1)
   Loop
    (cond
      [(null ranlist) (return nil)]
      [t
         (setq r1 (car ranlist))
         (putprop r1 (cons ran (get r1 'succ)) 'succ)
         (setq ranlist (cdr ranlist))
         (go Loop)]
]
(defun mkC1 ()
;this fcn operates on cralis to form the 'C1 set
;C1 consists of pairs of ranges which may be in parallel
  (prog (r1 crtmp)
    (setq C1 nil)
    (mktmpsux);set up temporary lists of successors
     ;now, starting with the last range (first element of cralis)
     ;walk through the rest of the list
    (setq crtmp cralis)
   Loop
    (mkdots 1)
    (cond
     [(null crtmp) (return nil)]
     [t
       (setq r1 (car crtmp))
       (setq crtmp (cdr crtmp))
       (mkr1s r1 crtmp)
       (go Loop)])
]
```

```
(defun mkr1s (ran ranlis)
;this fcn constructs all C1 elements that have ran in them
  (prog ()
   Loop
    (cond
      [(null ranlis) (return nil)]
      [t
        (cond
           [(eq (car (get (car ranlis) 'tsuc)) ran)
            (putprop (car ranlis) (cdr (get (car ranlis) 'tsuc)) 'tsuc)]
           [t
            (setq C1 (cons (list ran (car ranlis)) C1))])
        (setq ranlis (cdr ranlis))
        (go Loop)]
]
(defun mktmpsux ()
;this fcn copies the succ lists of each range in cralis
;so they can be used destructively
  (prog (r1 crtmp)
    (setq crtmp cralis)
   Loop1
    (cond
      [(null crtmp) (return nil)]
      [t
        (setq r1 (car crtmp))
        (putprop r1 (get r1 'succ) 'tsuc)
        (setq crtmp (cdr crtmp))
        (go Loop1)]
]
(defun mkC2 ()
;this fcn constructs the C2 subset of C1, ie the set of pairs of
;superranges that are actually in conflict with one another.
  (prog (ranpair c1tmp)
     (setq c1tmp C1)
     (setq C2 nil)
     (setupW)
   Loop
     (mkdots 1)
     (cond
       ((null c1tmp) (return 'done))
       [t
         (setq ranpair (car c1tmp))
         (setq c1tmp (cdr c1tmp))
         (cond
           [(confp ranpair)
             (setq C2 (cons ranpair C2))])
         (go Loop)]
]

(defun nooverlap (ranpair)
;this fcn retruns t iff it can be guaranteed that abs. time overlap
;is definitely avoided
 (cond (lattice$bool
```

```
  (or (greaterp
       (get (car (get (get (car ranpair) 'rprc) 'rsux)) 'EB)
       ;i.e. the EB of the range that succeeds the pt that precedes the
       ;first element of the ranpair in question
       (get (car (get (get (cadr ranpair) 'rsuc) 'rsux)) 'LB))
       ;i.e. the LB of the range that succeeds the point that succeeds
       ;the second element of the ranpair
      (greaterp
       (get (car (get (get (cadr ranpair) 'rprc) 'rsux)) 'EB)
       ;i.e. the EB of the range that succeeds the pt that precedes the
       ;second element of the ranpair in question
       (get (car (get (get (car ranpair) 'rsuc) 'rsux)) 'LB))))))
       ;i.e. the LB of the range that succeeds the point that succeeds
       ;the first element of the ranpair
       ;this is damnably confusing. Sorry, I don't like it either.
       ;Consult nooverlap2 for a clearer explication based on another
       ;data structure.

(defun nooverlap2 (r1 r2)
;this fcn does the same as nooverlap but it operates on uncrush'd ranges
;recall EB is earliest beginning, LB latest beginning, EE earliest end,
;and LE latest end.
  (or
    (greaterp (get r1 'EB) (get r2 'LE))
    (greaterp (get r2 'EB) (get r1 'LE))))

(defun confp (ranpair)
;this fcn returns t if (car ranpair) and (cadr ranpair) are in true conflict
  (prog (sq)
    (setq sq (sequence 4));this is the sq-th confp test.
    ;if they have the same source pt, return t if its a afork
    [cond
      [(eq (get (car ranpair) 'rprc) (get (cadr ranpair) 'rprc))
         (return (eq (get (get (car ranpair) 'rprc) 'ptype) 'afork))]]
    ;see if they don't overlap in time
    [cond ((nooverlap ranpair) (return nil))]
    (setupUCW (car ranpair) (cadr ranpair))
    (putprop (car ranpair) 'a 'flagg)
    (putprop (cadr ranpair) 'b 'flagg)
    (putprop (car ranpair) sq 'c2mno)
    (putprop (cadr ranpair) sq 'c2mno)
   Loop
    (cond
       [(car C)
           ;if C is a death pt or if its the only pt then return
         (cond ((death (car C)) (return t)))
         (setq p (car C)) (setq C (cdr C))
         ;otherwise remove it from C
         (hopf p)
         ;and trace back along an in-arc
         (go Loop)
       ]
       [(car U)
```

```
        ;there are unresolved joins, which look like forks because we
        ;are transitting backwards, remember.
          (cond ((allmarkedq (car U) 'c2mno 'PreSupRan sq)
              ;if all its in-arcs have been expanded out of, forget about it
              (setq U (cdr U)))
            (t (setq C (cons (car U) C))))
            ;expand out from the pt
          (go Loop)
        ]
        [(isaW) ;if there is no U or C, there may be an unresolved join
           ;which, under these circumstances, can never be resolved. So,
           ;take the latest of these and make it the hoppable pt.
          (setq C (cons (deflictW) C))
          (go Loop)
        ]
        ;if its imposs to expand then the no. of source pts must be 1
        [t (return (greaterp (length V) 1))])
]

(defun setupUCW (r1 r2)
;this fcn presumes that the two ranges do not have a common source pt
;it adds the src pts to either U or C or W depending
    (setq U nil C nil V nil)
    (mkW);init the W array
    (cond
      [(joinp (get r1 'rprc)) (setq U (cons (get r1 'rprc) U))]
      [(forkp (get r1 'rprc)) (inflictW (get r1 'rprc))]
      [t (setq C (cons (get r1 'rprc) C))])
    (cond
      [(joinp (get r2 'rprc)) (setq U (cons (get r2 'rprc) U))]
      [(forkp (get r2 'rprc)) (inflictW (get r2 'rprc))]
      [t (setq C (cons (get r2 'rprc) C))])
]

(defun getran1 (pointname)
;gets the first unmarked element of 'rsux of pointname
;unmarked meaning 'sqno=nil
  (prog (L)
    (setq L (get pointname 'rsux))
   Loop
      (cond
        ((null L) (return nil))
        ((null (get (car L) 'sqno)) (return (car L)))
        (t (setq L (cdr L)) (go Loop))
]

(defun getran2 (pointname sno)
;gets the first unmarked 'PreSupRan of pointname
;unmarked meaning sno<>'c2mno
  (prog (L)
    (setq L (get pointname 'PreSupRan))
   Loop
      (cond
```

```
        ((null L) (return nil))
        ((neq (get (car L) 'c2mno) sno) (return (car L)))
        (t (setq L (cdr L)) (go Loop))
]


(defun marktheprogen (point)
;this fcn sets the 'flagg field of a point
  (prog (r L)
    (setq L (get point 'SucSupRan))
    (putprop point nil 'flagg)
   Loop
     (cond
       ((null L) (return nil))
       (t
         (setq r (car L)) (setq L (cdr L))
         (cond
           ((eq (refersq 4) (get r 'c2mno))
             (putprop point (motch point r) 'flagg) (go Loop))
           (t (go Loop)))
]

(defun motch (point range)
;this fcn returns a 'flagg value of 'a, 'b, or 'ab depending on
;what it sees as the flaggs of its input things
  (prog (x y r)
   (setq x (get point 'flagg) y (get range 'flagg))
   (setq r (cond
     ((null x) y)
     ((null y) x)
     ((neq x y) 'ab)
     (t x)))
   (return r);this could be cleaner
]

(defun death (point)
;this fcn returns t if:
 ;the pt is a afork pt and
 ;two of its outarcs are marked ab or one is marked a and another b
  (cond
    [(eq (get point 'ptype) 'afork)
      ;unless its an afork it cannot be a death point
      (mixdp point)]
      ;mixdp is the afork-specific death function
    [t nil]
]

(defun mixdp (point)
;this fcn determines if an afork has inappropriate outarcs
;for a given member of C1.
  (prog (r L flagg)
    (setq L (get point 'SucSupRan))
      ;L is the outarcs of point, flagg is a temporary flag value
```

```
    (setq flagg nil)
  Loop
    (cond
      ((null L) (return nil))
      ;if the outarc set is nil and no conflict has been detected there is no
      ;conflict
      (t
        (setq r (car L)) (setq L (cdr L));look at the next element of L
        (cond
          ((and (null flagg) (eq (refersq 4) (get r 'c2mno)))
          ;set flagg equal to the 'flagg field of the first current arc
            (setq flagg (get r 'flagg)) (go Loop))
          ((null flagg) (go Loop))
          ;arc is not current so ignore it
          ((eq (refersq 4) (get r 'c2mno))
          ;arc is current and flagg is set so check
            (cond
              ((or (and (eq flagg 'ab) (eq (get r 'flagg) 'ab))
                   (neq flagg (get r 'flagg)))
                   ;either two 'ab arcs or an 'a and a 'b is death
                (return t))
              (t (go Loop)))))
              ;two 'as or two 'bs is not death, but keep checking, who knows?
]


(defun hopf (point)
;this fcn hops from a pt along an in-arc to its predecessor pt.
;it is rather similar to hopc.
  (prog (ran sq)
    (setq sq (refersq 4))
    (setq ran (getran2 point sq))
    (putprop point sq 'c2mno)
        ;chd this fm p to point; reordered so cond is after marktheprogen
    (marktheprogen point)
    ;check to see is the point a global source; if it is add its flagg to V
    [cond ((null ran) (setq V (insert (get point 'flagg) V nil t))
             (return nil))]
    (putprop ran sq 'c2mno)
    (putprop ran (get point 'flagg) 'flagg)
    ;see what kind of pt its src pt is
    (cond
      [(simplep (get ran 'rprc))
         ;NB this should not happen in ordinary cases, just sinks and sources.
        (setq C (appendl C (get ran 'rprc)))]
      [(joinp (get ran 'rprc))
         ;if it is a join pt. then put it into U for unresolved
         ;a join pt cannot be a global source and when it is reached
         ;it cannot already have marked in-arcs
        (setq U (appendl U (get ran 'rprc)))]
      ;if it is a fork pt then two possible cases exist.
      ;either all of its out-arcs are marked, or they are not.
      ;if they are, it is a vanilla candidate for expansion as would
```

```
      ;be a simple pt.
      ;If they are not then put it into W which will be expanded only
      ;if there are no other expandable points
      [t (inflictW (get ran 'rprc))]
]

(defun ckbn ()
;this fcn walks thru molist checking the bindings
;to see if for any M e molist there exist B1 & B2 s.t. (R1 R2) e C2
;if so then there is a possible collision
  (prog (M tmo)
    (setq tmo molist)
    (setq C3 nil collbool nil);global data strux
  Loop
   (cond
     [tmo
       (setq M (car tmo))
       (setq tmo (cdr tmo))
       (CKB M) (go Loop)]
     [t (return 'done)])
]
(defun CKB (smodul)
;this fcn checks for modul, if there exists B1 & B2 st {R1 R2} e C2
;if so it enters {B1 B2} on C3
  (prog (b blis)
    (setq blis (get smodul 'brex))
  Loop
    (cond
      [blis
        (setq b (car blis))
        (setq blis (cdr blis))
        (BCK b blis) (go Loop)]
      [t (return nil)])
]
(defun BCK (bindi bindl)
;this fcn checks for conflicts between bindi and elements of bindl
;and reports conflicts
  (prog (bindj)
    (mkdots 1)
  Loop
    (cond
      [bindl
        (setq bindj (car bindl))
        (setq bindl (cdr bindl))
        (cond
          ((collision bindi bindj)
            (reportcoll bindi bindj)))
        (go Loop)]
      [t (return nil)])
]
(defun reportcoll (b1 b2)
;this fcn reports collisions
;it detects them by table lookup
```

```
    (setq C3 (cons (list b1 b2) C3))
    (setq collbool t)
  ]
  (defun collision (b1 b2)
  ;detects a collision
    (prog (x y p q r)
     (setq p (caddr b1) q (caddr b2))
     (cond ((and lattice$bool (nooverlap2 p q)) (return nil)))
     (setq x (get p 'eqset) y (get q 'eqset))
     (setq r (eq p q))
     (setq r (or r
        (and (neq x y)
        (cond
          [(greaterp (get x 'sqno) (get y 'sqno))
           (member (list x y) C2)]
          [t (member (list y x) C2)]))))
     (return r)
  ]

  (defun setupW ()
  ;this fcn constructs the W array and initializes it
    (prog (sz pls)
      (setq sz  (refersq 5) pls polist)
      (*array 'W t sz)
      (setq Wcount$ 0)
    Loop
      (cond ((null pls) (return nil))
       (t
         (cond
           ((forkp (car pls))
             (store (W (get (car pls) 'wno)) (list 0 (car pls)))))
         (setq pls (cdr pls)) (go Loop))))
  )

  (defun mkW ()
  ;this fcn initializes the W array
    (setq Wcount$ 0 Wptr$ (refersq 5))
  )
  (defun isaW () (greaterp Wcount$ 0));is W nonempty?

  (defun inflictW (E)
  ;this fcn inserts E into W in the slot of its wno
     (cond
       ((neq (refersq 4) (car (W (get E 'wno))))
         (setq Wcount$ (+ 1 Wcount$))
         (store (W (get E 'wno)) (list (refersq 4) E))))
     (cond
       ((greaterp (get E 'wno) Wptr$) (setq Wptr$ (get E 'wno))))
  )

  (defun deflictW ()
  ;this fcn gets the last active element of W and returns it
    (prog (tE)
```

```
 Loop
   (cond
     ((lessp Wptr$ 0) (return nil))
     ((eq (refersq 4) (car (W Wptr$)))
       (setq tE (cadr (W Wptr$)))
       (store (W Wptr$) (list 0 tE))
       (setq Wcount$ (- Wcount$ 1))
       (setq Wptr$ (- Wptr$ 1))
       (return tE))
     (t (setq Wptr$ (- Wptr$ 1)) (go Loop))))
)
;that's all the defns
;now to run the program
;just call R
(R)
```

Appendix 3

Documentation and Program Listing for Clocking Scheme Synthesis

A Guide to CSSP

(Clocking Scheme Synthesis Package)

Version 1.1

By Nohbyung Park

31 October 1984

Table of Contents

List of Figures

1 INTRODUCTION

This document contains a brief introduction to the CSSP (Clocking Scheme Synthesis Package) which implements the clocking scheme synthesis algorithms developed by [Park 84]. We assume that the readers are familiar with the clocking scheme synthesis algorithms described in [Park 84].

Section 2 describes the input requirements of the system. Section 3 describes the operations of major routines of the system and how to use them. In Section 4, the global variables containing the current results of the stage partitioning and critical path analysis are described. Section 5 shows a sample run of the CSSP.

2 INPUT REQUIREMENTS OF THE CSSP

The description of the input MEG to be analyzed is stored in two separate files, a *node description file* and an *edge description file.* Section 2.1 describes the node description file and Section 2.2 describes the edge description file. In Section 2.3, an example of a MEG and its node and edge descriptions are given. Besides the node and edge descriptions, the user can specify the default delay times, Dss and Dsp [Hafer 83], of the stage latches to be used. If not explicitly specified, they are set to zero.

2.1 Node Input File

A node input file contains a "LISP" list of the description of the nodes of the MEG to be analyzed. Each node description itself is again a list of a node name and a module propagation delay (MPD).

```
NodeList ::= (<Node> <Node> {<Node>}*)

             <Node> ::= (<NodeName><MPD>)

             <NodeName> ::= a string of alphanumerics.

             <MPD> ::= an integer (nsec in default).
```

The <NodeName> of each node must be unique and can be up to 80 characters

long. <MPD> is the worst case (longest) delay time of the corresponding functional module.

2.2 Edge Input File

An edge input file contains a "LISP" list of the descriptions of the edges of the MEG to be analyzed. An edge description is a list of six-tuple of edge name, source node name, sink node name, bitwidth, the number of delay registers and the number of bypass registers on the edge.

```
EdgeList ::= (<EdgeId><SourceNode><SinkNode>
                    <BitWidth><#DlyReg> [<#ByReg>])

        <EdgeId> ::= a string of alphanumerics

        <SourceNode> ::= <NodeName>

        <SinkNode> ::= <NodeName>

        <BitWidth> ::= integer

        <#DlyReg> ::= integer | nil

        <#ByReg> ::= integer | nil
```

More than one edge may have the same <EdgeId> if the edges are physically connected to the same output port(s). Delay registers are those which are used either to read or to write but *not to write and then read* during one micro cycle or major clock cycle. Bypass registers are those which are written and then read during one micro cycle or major clock cycle.

2.3 An Example

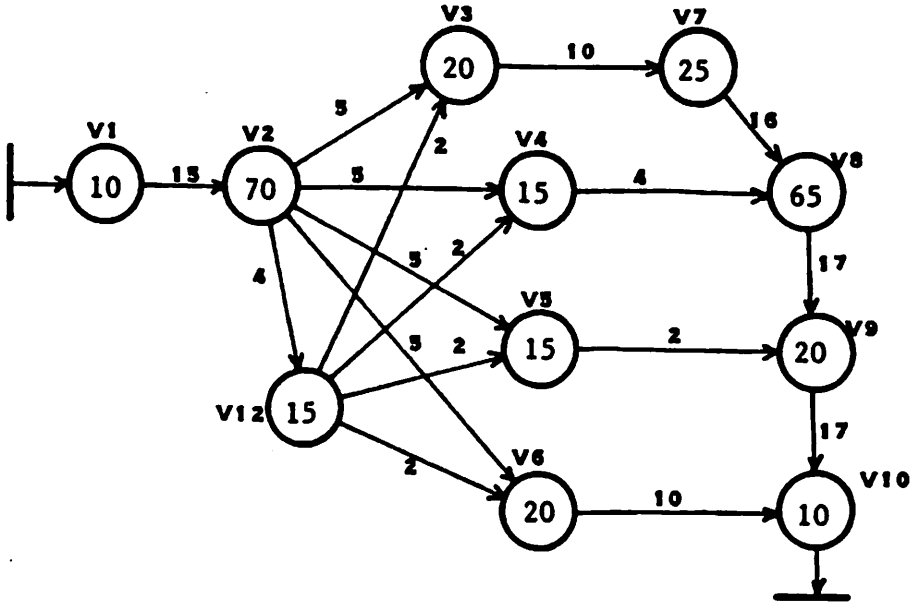


Figure 1: An Example MEG

```
( (v1 10) (v2 70) (v3 20) (v4 15) (v5 15)
  (v6 20) (v7 25) (v8 65) (v9 20) (v10 10)
  (v12 15) )
```

Figure 2: The Node Description List for the MEG of Fig. 1

```
((PA     v1  v2 15 0 0) (IF1    v2  v3  5 0 0)
 (IF2    v2  v4  5 0 0) (IF3    v2  v5  5 0 0)
 (IF4    v2  v6  5 0 0) (OP     v2 v12  4 0 0)
 (SRC   v12  v3  2 0 0) (ALU   v12  v4  2 0 0)
 (SR    v12  v5  2 0 0) (DST   v12  v6  2 0 0)
 (CS1    v3  v7 10 0 0) (CS2    v4  v8  4 0 0)
 (CS3    v5  v9  2 0 0) (CS4    v6 v10 10 0 0)
 (INPUT  v7  v8 16 0 0) (RESULT v8  v9 17 0 0)
 (OUTPUT v9 v10 16 0 0))
```

Figure 3: The Edge Description List for the MEG of Fig. 1

2.4 Other Input Considerations

Besides the node and edge descriptions, the user may need to preset the Dss (the maximum input setup time) and Dsp (the maximum storage propagation time) of the storage elements.

In default, these values are set to zero. Also, the user can specify and/or modify these values when he/she calls some of the major procedures of the package. We will discuss this in the following sections in detail.

3 OPERATIONS OF THE CSSP

The CSSP consists of four major procedures, inputprocess, rkpart, opart and cp. We will first discuss each procedure and then discuss the overall operation of the CSSP. For more details of these procedures, the reader is urged to refer the source code.

3.1 The Procedure inputprocess

Command Format: (*inputprocess* 'nodefilename 'edgefilename)

where *nodefilename* is the file containing the node description list and *edgefilename* containing the edge description list.

This procedure reads in the node and edge description files described in the previous chapter and build the MEG to be analyzed. It also initializes all the global buffers shared by the other three procedures. This procedure must be called and executed whenever a new MEG is to be analyzed.

3.2 The Procedure rkpart

Command Format: (*rkpart* lmax dss dsp)

where *lmax* is the maximum stage time allowed, *dss* is the storage set up time and *dsp* is the storage propagation time of the stage latches.

This procedure partitions the MEG in such a way that the stage time of any stage is

not longer than the allowed maximum value (lmax) including the delay times of the stage latches (dss and/or dsp).

3.3 The Procedure opart

Command Format: (*opart* k)

where k is the desired number of stages.

This procedure partitions the MEG into the desired number of stages (k) while minimizing the longest stage time.

3.4 The Procedure cp

Command Format: (*cp* dss dsp)

where *dss* and *dsp* are the storage delay times of the stage latches as described before. The dss and dsp passed to this procedure do not change the global dss and dsp values used by the procedure *opart*.

3.5 Overall Operation Considerations of the CSSP

All four procedures are independent of each other and can be called in any order. However, some global variables may have different values according to the order of calling the procedures.

The standard procedure for using this CSSP can be summarized as follows:

1. Call *inputprocess* and process the input files.

2. Set the desired values of Dss and Dsp. (The default value of these variables is zero and can be altered at any time)

3. Call *rkpart* or *opart* to carry out stage partitioning depending on the objective function.

   - New values of Dss and Dsp can be passed to *rkpart* without altering them permanently.

   - The procedures, kpart and opart, can be called as many times as

needed. They access exactly the same variables (in fact, opart calls kpart iteratively) during execution. Therefore, any result to be saved must be printed out before calling any other procedure including itself.

4. The procedure *cp* can be called at any time to detect the critical path in the MEG either before or after any stage partitioning. Dss and Dsp can be passed as local variables to this procedure.

4 GLOBAL VARIABLES

Some important variables are declared as global variables. Although this is considered to be dangerous in LISP programming, it is convenient when the user wants to examine and print out the results selectively. The global variables are summarized below.

nodelist	A list. Same as the input node description list.
mpd	A list of the module propagation delays.
nodenamelist	A list of the NodeName's only.
numofnodes	An integer. The number of the nodes.
edgelist	A list. Same as the input edge description list.
BWList	A list. Contains the bitwidth of the edges.
numofedges	An integer. The number of edges.
adjm	The Adjacency matrix for the MEG.
incdm	The incidence matrix (edges) showing the incoming/outgoing edges to/from each node. This matrix has two columns with as many rows as the number of the nodes in the MEG. Column 0/1 contains a list of indices of all the incoming/outgoing arcs to/from the *row*-th node.
neighbors	A matrix showing neighboring nodes (parents and children) of each node. This matrix has two columns with as many rows as the number of the nodes. Column 0/1 contains a list of indices of all the parents/children of the *row*-th node.

cutsets	A matrix. Stores the current edge cutsets for the locations of stage latches either after calling *rkpart* or *opart*.
spd	A matrix. Stores the stage times for the current stage partitioning.
cpmarkn	A matrix. Used to mark the critical path nodes.
cpmarke	A matrix. Used to mark the critical path edges.

All these global variables can be accessed and printed by standard LISP printing commands. Especially for the matrices, the functions *apr3* and *apl3* are provided in this package. The function *apr3* prints a fixnum (integer) type matrix and *apl3* prints any other types of matrix.

5 A SAMPLE RUN OF THE CSSP

This section shows a sample run of the CSSP package for the example MEG shown in Section 2.3.

```
Script started on Tue Oct 23 02:26:49 1984
Noh[1] lisp
Franz Lisp, Opus 38.79
-> (load 'cssp.l)
[load cssp.l]
t
-> (inputprocess 'nl0 'el0)

   *****   MODULE  LIST   *****

   (MODULE_NAME MPD<nsec>)

    m  0  =  (u1 10)
    m  1  =  (u2 70)
    m  2  =  (u3 20)
    m  3  =  (u4 15)
    m  4  =  (u5 15)
    m  5  =  (u6 20)
    m  6  =  (u7 25)
    m  7  =  (u8 65)
    m  8  =  (u9 20)
    m  9  =  (u10 10)
    m 10  =  (u11 15)


   *****   EDGE  LIST   *****

(EDGE_NAME SRC SINK BITWIDTH DR BR)

    e  0  =(PA u1 u2 15 0 0)
    e  1  =(IF1 u2 u3 5 0 0)
    e  2  =(IF2 u2 u4 5 0 0)
    e  3  =(IF3 u2 u5 5 0 0)
    e  4  =(IF4 u2 u6 5 0 0)
    e  5  =(OP u2 u11 4 0 0)
    e  6  =(SRC u11 u3 2 0 0)
    e  7  =(ALU u11 u4 2 0 0)
    e  8  =(SR u11 u5 2 0 0)
    e  9  =(DST u11 u6 2 0 0)
    e 10  =(CS1 u3 u7 10 0 0)
```

```
    e 11  =(CS2 u4 u8 4 0 0)
    e 12  =(CS3 u5 u9 2 0 0)
    e 13  =(CS4 u6 u10 10 0 0)
    e 14  =(INPUT u7 u8 16 0 0)
    e 15  =(RESULT u8 u9 17 0 0)
    e 16  =(OUTPUT u9 u10 16 0 0)

"Input Lists Processed Successfully!"

t
-> (apr3 'adjm)

INDEX   0   1   2   3   4   5   6   7   8   9  10

   0    0   1   0   0   0   0   0   0   0   0   0
   1    0   0   1   1   1   1   0   0   0   0   1
   2    0   0   0   0   0   0   1   0   0   0   0
   3    0   0   0   0   0   0   0   1   0   0   0
   4    0   0   0   0   0   0   0   0   1   0   0
   5    0   0   0   0   0   0   0   0   0   1   0
   6    0   0   0   0   0   0   0   1   0   0   0
   7    0   0   0   0   0   0   0   0   1   0   0
   8    0   0   0   0   0   0   0   0   0   1   0
   9    0   0   0   0   0   0   0   0   0   0   0
  10    0   0   1   1   1   1   0   0   0   0   0
t
-> (apl3 'incdm)

ROW
  0   nil                    (0)
  1   (0)                    (1 2 3 4 5)
  2   (1 6)                  (10)
  3   (2 7)                  (11)
  4   (3 8)                  (12)
  5   (4 9)                  (13)
  6   (10)                   (14)
  7   (11 14)                (15)
  8   (12 15)                (16)
  9   (13 16)                nil
 10   (5)                    (6 7 8 9)
t
```

```
-> (apl3 'neighbor)

ROW
  0   nil                    (1)
  1   (0)                    (2 3 4 5 10)
  2   (1 10)                 (6)
  3   (1 10)                 (7)
  4   (1 10)                 (8)
  5   (1 10)                 (9)
  6   (2)                    (7)
  7   (3 6)                  (8)
  8   (4 7)                  (9)
  9   (5 8)                  nil
 10   (1)                    (2 3 4 5)
t

-> (rkpart 100 10 10)

  **********  EDGE-CUTSETs  **********

  cutset  1  =(1 2 3 4 5)
  cutset  2  =(12 13 11 14)
  cutset  3  =(13 12 15)


 *****  STAGE-PROPAGATION-DELAY  *****

    stage-delay  1  =90
    stage-delay  2  =80
    stage-delay  3  =85
    stage-delay  4  =40
4

-> (setq dss 5)
5
-> (setq dsp 10)
10
```

```
-> (opart 3)

"Realizable Stage Times are:"

(85 95 100 105 115 120 125 130 140 145 150 160 165 210 230 235)


"Enter rkpart with stage time:" 130  --->  k =  2

"Enter rkpart with stage time:" 105  --->  k =  3

"Enter rkpart with stage time:"  95  --->  k =  4

"Enter rkpart with stage time:" 100  --->  k =  4


"Enter rkpart with stage time:" 105

  **********  EDGE-CUTSETs  **********

  cutset  1  =(4 9 3 8 2 7 1 6)
  cutset  2  =(12 13 11 14)


 *****  STAGE-PROPAGATION-DELAY  *****

    stage-delay  1  =100
    stage-delay  2  =60
    stage-delay  3  =105

 dmax= 105

t
-> (exit)
Noh[2] ^D
script done on Tue Oct 23 02:33:17 1984
```

References

[Hafer 83] Hafer, L., and Parker, A.
A Formal Method for the Specification Analysis, and Design of Register-Transfer Level Digital Logic.
IEEE Transactions on Computer-Aided Design CAD-2(1), January, 1983.

[Park 84] Park N. and Parker, A.
Synthesis of Optimal Clocking Scheme for Digital Systems.
Technical Report DISC/84-1, Dept. of EE-Systems, University of Southern California, May, 1984.

Program Listings

```
;*************************************************************

; COMMON FUNCTIONS LIBRARY

;       This library contains all the common functions that
;       are called by more than one procedures. Detailed
;       description of each function is attached to the
;       source code as comments.

;       1. apr3          prints out a fixnum type matrix.

;       2. apl3          prints out a t type matrix.

;       3. zerocols      returns all the indices of zero
;                        columns of a matrix.

;       4. colsum        returns sum of a column of a
;                        matrix.

;       5. glabel        labels the nodes a directed acyclic
;                        graph with their depth and height.
;                        Also, the nodes with same depth/
;                        height are grouped.

;       6. zerorows      returns all the indices of zero
;                        rows of a matrix.

;       7. rowsum        returns sum of a row of a matrix.

;       8. fminlmax      returns the minimum possible stage
;                        time of a MEG.

;       9. ienum         enumerate all the possible stage
;                        times of a MEG.

;*************************************************************


(defun apr3 (fixnum_matrix)

;       Parameter: The name of the fixnum-type matrix
;              to be printed.

;       Description : This function prints out a fixnum
;              type matrix on the standard output device.
;              Each cell of the matrix is printed in a 4-
;              digit slot including the space between the
;              cells.
```



```
        (prog (apx apy p3x p3y)

                ; error return if not an array

                (cond ((not (arrayp fixnum_type))
                       (return "ERROR - Not an array")))

                ; get the dimensions of the matrix
                ; apx-rows ; apy-columns

                (setq apx (cadr (arraydims fixnum_matrix)))
                (setq apy (caddr (arraydims fixnum_matrix)))

                (setq p3y 0)
                (cprintf '%5s 'INDEX)
        loopap
                ; print column indices.

                (cprintf '%4d p3y)
                (setq p3y (add1 p3y))
                (cond ((not (eq p3y apy))(go loopap)))

                (terpr)
                (terpr)
                (setq p3x 0)
                (setq apy (sub1 apy))
        ap3loop1
                (cond ( (eq apx p3x) (return t)))
                (setq p3y 0)
                (cprintf '%4d p3x)
                (cprintf '%1s (ascii 32))

                ; print a row.
        ap3loop2
                (cprintf '%4d (fixnum_matrix p3x p3y))
                (cond ((not (eq apy p3y))
                       (setq p3y (add1 p3y)) (go ap3loop2)))
                (setq p3x (add1 p3x))
                (terpr)
                (go ap3loop1)
        )
)
```

```
(defun apl3 (t_matrix)

;
;       Parameter: The name of the t-type matrix to be printed.
;
;       Description : This function prints out a t type
;               matrix on the standard output device.
;               Each cell of the matrix is printed in a 25-
;               column slot including the space between the
;               cells.
;
;               If the array to be printed has more than 3
;               columns or any element with more than 25
;               characters, the printout may not be readable.
;


        (prog (apx apy p3x p3y)

                ; error return if not an array

                (cond ((not (arrayp fixnum_type))
                       (return "ERROR - Not an array")))

                ; get the dimensions of the matrix
                ; apx-rows ; apy-columns

                (setq apx (cadr (arraydims t_matrix)))
                (setq apy (caddr (arraydims t_matrix)))

                (setq p3y 0)
                (terpr)
                (cprintf '%3s 'ROW)
                (terpr)
                (setq p3x 0)
                (setq apy (sub1 apy))
        ap3loop1
                (cond ( (eq apx p3x) (return t)))
                (setq p3y 0)
                (cprintf '%3d p3x)
                (setq indent 8)
        ap3loop2
                (pp-form (t_matrix p3x p3y) 'dummy indent)
                (cond ((not (eq apy p3y))
                       (setq p3y (add1 p3y))
                       (setq indent (plus indent 25))
                       (go ap3loop2)
                      )
                )
                (setq p3x (add1 p3x))
                (terpr)
                (go ap3loop1)]
```



```
(defun zerocols (name)

;
;       Parameter: The name of the matrix to be checked.
;
;       Description: This routine returns all the column
;               indices of the given matrix whose sum is
;               zero.
;


        (prog (cntr buf)

                ; get the number of columns
                (setq cntr (caddr (arraydims name)))

                (setq buf '())
        loop
                (setq cntr (sub1 cntr))
                (cond ( (zerop (colsum name cntr))
                        (setq buf (cons cntr buf))))
                (cond ( (zerop cntr) (return buf)))
                (go loop)
        )
)


(defun colsum (name ix)

;
;       Parameter: The name of the matrix <name> and the
;               column index <ix> to be checked.
;
;       Description: This routine returns the sum of the
;               specified column of the given matrix.
;

        (prog (sum iy)
                (setq sum 0)

                ; get the number of rows
                (setq iy (cadr (arraydims name)))
        loop
                (setq iy (sub1 iy))
                (setq sum (add sum (name iy ix)))
                (cond ((zerop iy)(return sum)))
                (go loop)
        )
)
```

```
(defun glabel ()

;
;       Parameter: None.
;
;       Description: This function labels the nodes of the
;               input MEG (processed by the "inputprocess"
;               procedure) with their depth and height.
;               Also, the nodes with the same depth/height
;               are grouped together.
;
;               Depth of nodes are stored in the array nmat2
;               and height of nodes are stored in the array
;               nmat3. The depthmat/heightmat are containing
;               the nodes with the same depth/height.
;


        (flabel)
        (blabel)
        (gen_depth_height)
)


(defun flabel ()

;
;       Description: This function labels the nodes of the
;               input MEG with their depth. The depth of a
;               node is the maximum number of nodes in any
;               path from any root to the node.
;

        (prog (shead nhead lcnt)

                ; nmat2 is used as temporary buffer to store
                ; and pass to gen_depth_matrix the depth of
                ; the nodes.

                (fillarray 'nmat2 '(0))
                (setq lcnt 0)
                (setq nhead (zerocols 'adjm))
        loop
                (cond ( (zerop (length nhead)) (return t)))
                (setq shead nhead)
                (setq nhead '())
                (mapcar '(lambda (node)
                        (store (nmat2 node 0) lcnt)
                        (setq nhead (append (neighbor node 1) nhead)))
                 shead)
                (setq lcnt (add1 lcnt))
                (go loop)]
```



```
(defun blabel ()

;
;       Description: This function labels the nodes of the
;               input MEG with their height. The height of
;               a node is the maxinum number of nodes in
;               any path from the node to any leaf node.
;

        (prog (shead nhead lcnt)

                ; nmat3 is used to store the height of the
                ; height of the nodes.

                (fillarray 'nmat3 '(0))
                (setq lcnt 0)
                (setq nhead (zerorows 'adjm))
        loop
                (cond ( (zerop (length nhead)) (return t)))
                (setq shead nhead)
                (setq nhead '())
                (mapcar '(lambda (node)
                        (store (nmat3 node 0) lcnt)
                        (setq nhead (append (neighbor node 0) nhead)))
                 shead)
                (setq lcnt (add1 lcnt))
                (go loop)]


(defun zerorows (name)

;
;       Description: This function returns a list of row
;               indices  whose sum is zero.
;

        (prog (cntr buf)

                ; get the number of rows

                (setq cntr (cadr (arraydims name)))
                (setq buf '())
        loop
                (setq cntr (sub1 cntr))
                (cond ( (zerop (rowsum name cntr))
                        (setq buf (cons cntr buf))))
                (cond ( (zerop cntr) (return buf)))
                (go loop)]
```

```
(defun rowsum (name iy)

;
;       Parameter: The name <name> of the matrix and the
;               row index <iy>.
;
;       Description: This function returns the sum of the
;               row <iy>.
;

        (prog (sum ix)
                (setq sum 0)

                ; get the number of columns
                (setq ix (caddr (arraydims name)))
        loop
                (setq ix (sub1 ix))
                (setq sum (add sum (name iy ix)))
                (cond ((zerop ix)(return sum)))
                (go loop)]


(defun gen_depth_height ()

;
;       Description: This routine fills in the depth/height
;               matrices with the nodes having the same
;               depth/height.
;

        (prog (lcnt depthid heightid)

                ; depthmat and heightmat are declared as
                ; global variables.

                (array depthmat t numofnodes 1)
                (array heightmat t numofnodes 1)

                (setq lcnt numofnodes)
        loop
                (setq lcnt (sub1 lcnt))
                (setq depthid (nmat2 lcnt 0))
                (setq heightid (nmat3 lcnt 0))
                (store (depthmat depthid 0)
                       (cons lcnt (depthmat depthid 0)))
                (store (heightmat heightid 0)
                       (cons lcnt (heightmat heightid 0)))
                (cond ( (zerop lcnt) (return t)))
                (go loop)
        )
)
```



```
(defun fminlmax ()

;
;       Description: This function determines the minimum
;               realizable stage time for the input MEG.
;

    (prog (cntr wlist result)

        (setq wlist (mapcar '(lambda (x)
                                (plus x dss dsp)) mpd))
        (fillarray 'nmat1 wlist)
        (setq cntr numofnodes)
    loop
        (setq cntr (sub1 cntr))
        (cond ( (zerop (colsum 'adjm cntr))
                (store (nmat1 cntr 0)
                       (diff (nmat1 cntr 0) dsp))))

        ; check if weight is less than dss or dsp.

        (cond ( (zerop (rowsum 'adjm cntr))
                (store (nmat1 cntr 0)
                       (diff (nmat1 cntr 0) dss))))
        (cond ((not (zerop cntr)) (go loop)))
        (setq wlist (listarray 'nmat1))
        (return (apply 'max wlist))))


(defun ienum ()

;
;       Description: This routine finds all the realizable
;               stage times for the input MEG.
;

    (prog (ivlist minlmax)

        ; compute the minimum realizable stage time that
        ; is at least longer than the maximum MPD of the
        ; modules.

        (setq minlmax (fminlmax))

        ; find all the possible stage times
        (setq ivlist (ifind))

        ; remove all the intervals shorter than minlmax.
        (setq ivlist (iopt ivlist))
        (setq ivlist (reverse ivlist))
        (return ivlist)
))
```

```
(defun ifind ()

;
;       Description: This routine finds all the possible
;               stage times including those unrealizable.
;

    (prog (ncntr intvl)

        (setq intvl '())
        (setq ncntr 0)
    loop
        (nflabel (list ncntr))
        (ngdepth)
        (eigenw)
        (setq intvl (append intvl (listarray 'nmat2)))
        (setq ncntr (add1 ncntr))
        (cond ((eq ncntr numofnodes) (return intvl)))
        (go loop)
))


(defun nflabel (sset)

;
;       Parameter: A list of nodes <sset> as the starting
;               nodes (level 0).
;
;       Description: This function labels the nodes of the
;               input MEG with the depth the of the nodes
;               starting at the nodes in <sset>.
;
;       Global buffer used: nmat1 containing the node levels
;

    (prog (shead nhead lcnt DRList cntr nodeid)

        (setq DRList (mapcar 'caddddr edgelist))

        ; the level of any un-visited node is -1.

        (fillarray 'nmat1 '(-1))
        (setq lcnt 0)
        (setq nhead sset)
    loop
        (cond ((null nhead) (return t)))
        (setq shead nhead)
        (setq nhead '())
        (setq cntr (length shead))
    loop1
        (setq nodeid (nthelem cntr shead))
        (store (nmat1 nodeid 0)
               (max (nmat1 nodeid 0) lcnt))
```



```
        ; Put in nhead all the children who can be reached
        ; without passing any register

        (cond ( (not (null (incdm nodeid 1)))
                (mapcar
                 '(lambda (x)
                     (if   (zerop (nth x DRList))
                      then (setq nhead
                                 (append nhead
                                         (list (eval
                                                (caddr (nth x edgelist))))))))
                 (incdm nodeid 1))
              )
        )

        (setq cntr (sub1 cntr))
        (cond ((not (zerop cntr)) (go loop1)))

        (setq lcnt (add1 lcnt))
        (go loop)))


(defun ngdepth ()

;
;       Description: As input, levels of the nodes are
;               stored in nmat1. Output level-groups
;               will overwrite nmat1.
;

    (prog (levellist ngcnt levelid)

        (setq levellist (listarray 'nmat1))
        (fillarray 'nmat1 '(()))
        (setq ngcnt numofnodes)
    loop
        (setq levelid (nthelem ngcnt levellist))
        (setq ngcnt (sub1 ngcnt))
        (cond ( (not (eq levelid -1))
                (store (nmat1 levelid 0)
                (cons ngcnt (nmat1 levelid 0)))
              )
        )
        (cond ((zerop ngcnt) (return t)))
        (go loop)))
```

```
(defun eigenw ()

;
;       Description: This function takes depth groupes via
;               nmat1 as input, and computes and and stores
;               the accumulated maximum weights from any
;               root(s) to a node in nmat2.
;
;               Calls e1iw and fef.
;

    (prog (from1 to1 lcnt)

        (setq e11 '())
        (setq lcnt 0)
        (fillarray 'nmat2 '(0))
        (mapcar 'e1iw (nmat1 0 0))
    loop
        (setq from1 (nmat1 lcnt 0))
        (setq to1 (nmat1 (add1 lcnt) 0))
        (cond ((null to1) (return t)))
        (mapcar 'fef from1)
        (setq lcnt (add1 lcnt))
        (go loop)))


(defun e1iw (nodeid)

;
;       Description: This function initializes the weight
;               of the nodes before a new propagation of
;               the accumulated weights. Dss and/or Dsp
;               are also added if necessary.
;

    (prog (w)
        (setq w (nth nodeid mpd))
        ( if    (zerop (colsum 'adjm nodeid))
          then  (setq w (plus w dss))
          else  (setq w (plus w dsp dss))
        )
        (store (nmat2 nodeid 0) w)))


(defun fef (fathernode)

    (prog (father)
        (setq father fathernode)
        (mapcar 'fec to1)))
```



```
(defun fec (childnode)

    (prog (child w)

        (setq child childnode)
        (cond ((zerop (adjm father child)) (return t)))
        (setq w (plus (nmat2 father 0)
                      (nth child mpd)))
        (cond ((zerop (rowsum 'adjm child))
               (setq w (diff w dss))))
        (store (nmat2 child 0) w)))


(defun iopt (row1)

;
;       Parameter: A list of fixnum numbers <row1>.
;
;       Description: Remove any unrealizable stage times
;               and duplicate copies of the same stage time
;

    (prog (ilist buf1 lastw icnt)
        (setq ilist row1)
        (setq ilist (sort ilist 'lessp))
        (setq buf1 '())
        (setq lastw 0)
        (setq icnt 1)
    loop
        (setq num (nthelem icnt ilist))
        (setq icnt (add1 icnt))
        (cond ((eq num nil) (return buf1)))
        (cond ((lessp num minlmax) (go loop)))
        (cond ((not (eq num lastw))
               (setq lastw num)
               (setq buf1 (cons num buf1))
              )
        )
        (go loop)
    )
)
```

```
;*************************************************************
;
;       INPUT PROCESSING PROCEDURE
;
;               This procedure performs all the necessary
;               preparations for the stage partitioning
;               and clocking scheme synthesis procedures.
;
;       Main Sub-routines:
;
;               1. nodeprocess  reads in and translates the
;                               node input file <nodeinput>
;
;               2. edgeprocess  reads in and translates the
;                               edge input file <edgeinput>
;                               Also, constructs the MEG.
;
;               3. seties       counts the number of edges
;                               coming in to every node.
;
;               4. init_buffers initializes global buffer
;                               matrices.
;*************************************************************

(defun  inputprocess (nodeinput edgeinput)

    (prog ()

        (nodeprocess nodeinput)
        (edgeprocess edgeinput)
        (seties)
        (init_buffers)

        ; Set default dss and dsp.

        (setq dss 0)
        (setq dsp 0)

        (print "Input Lists Processed Successfully!")
        (terpr)
        (terpr)
        (return t)
    )
)
```



```
(defun nodeprocess (nodelistfilename)

;
;       Parameter: Node input file name <nodelistfilename>
;
;       Description: This routine creates a node list, an
;               MPD list, and a list of the node names only.
;               Also, an internal node index is attached to
;               each node. Finally, the all the nodes with
;               node indices is printed out.
;

        ; read in the node list file.
        (setq nodelist (read (infile nodelistfilename)))

        ; make a list of MPD's only.
        (setq mpd (mapcar 'cadr nodelist))

        ; the number of nodes is a global variable.
        (setq numofnodes (length nodelist))

        (setq nodenamelist (mapcar 'car nodelist))

        ; assign internal node indices to the nodes
        ; in the same order as they read in, starting
        ; from 0.
        (name2id nodenamelist)
        (printnodelist))


(defun name2id (namelist)

;
;       Description: This function assigns each node an
;                internal node index number.
;

        (prog   (command namebuf cntr idbuf)

                (setq namebuf (reverse namelist))
                (setq cntr (length namebuf))
                (setq command '(setq nodename cntr))
        loop
                (cond ((zerop cntr) (return t)))
                (setq cntr (subi cntr))
                (setq idbuf (car namebuf))
                (setq namebuf (cdr namebuf))
                (eval (subst idbuf 'nodename command))
                (go loop)))
```

```
(defun printnodelist ()

;
;       Descripttion: This routine prints out the nodes
;               description with the internal node id's.
;

        (prog (pncntr)

                (setq pncntr 0)
                (terpr)

                (cprintf '%8s '*****)
                (cprintf '%9s 'MODULE)
                (cprintf '%6s 'LIST)
                (cprintf '%8s '*****)
                (terpr)
                (terpr)
                (cprintf '%3s (ascii 32))
                (print '(MODULE_NAME  MPD<nsec>))
                (terpr)
        pnloop
                (terpr)
                (cprintf '%5s 'n)
                (cprintf '%3d pncntr)
                (cprintf '%3s '=)
                (cprintf '%2s (ascii 32))
                (print (nth pncntr nodelist))
                (setq pncntr (add1 pncntr))
                (cond ( (eq pncntr numofnodes)(terpr)(return t)))
                (go pnloop)))
```



```
(defun edgeprocess (edgelistfilename)

;
;       Parameter: Input edge file name <edgelistfilename>.
;
;       Description: Process input <edgelistfile> and set up
;               the following global variables:
;
;               1. Adjacency matrix
;               2. Incidence (edges) set matrix
;               3. Neighbor (nodes) set matrix
;               4. A list of bitwidth of the edges
;

        (prog   (ebuf cntr from to)

                (setq edgelist (read (infile edgelistfilename)))
                (setq numofedges (length edgelist))
                (setq cntr numofedges)

                (array incdm t numofnodes 2)
                (array neighbor t numofnodes 2)
                (array adjm fixnum numofnodes numofnodes)

                (setq BWList (mapcar 'cadddr edgelist))
        loop
                (cond ((zerop cntr)(pelist)(terpr)(return t)))

                (setq ebuf (nthelem cntr edgelist))
                (setq from (eval (cadr ebuf)))
                (setq to (eval (caddr ebuf)))
                (store (adjm from to) 1)
                (setq cntr (sub1 cntr))

                (store (incdm from 1) (cons cntr (incdm from 1)))
                (store (incdm to 0) (cons cntr (incdm to 0)))
                (store (neighbor from 1) (cons to (neighbor from 1)))
                (store (neighbor to 0) (cons from (neighbor to 0)))

                (go loop)
))


(defun init_buffers ()

        (array nmat1 t numofnodes 1)
        (array nmat2 t numofnodes 1)
        (array nmat3 t numofnodes 1)
        (array emat1 t numofedges 1)
)
```

```
(defun pelist ()

;
;       Description: Print out the edges with their internal
;               index numbers.
;

        (prog (pecntr)

                (setq pecntr 0)
                (terpr)
                (cprintf '%9s '*****)
                (cprintf '%7s 'EDGE)
                (cprintf '%8s 'LIST)
                (cprintf '%8s '*****)
                (terpr)
                (terpr)
                (print '(EDGE_NAME SRC SINK BITWIDTH DR BR))
                (terpr)
        pnloop
                (terpr)
                (cprintf '%5s 'e)
                (cprintf '%3d pecntr)
                (cprintf '%3s '=)
                (print (nth pecntr edgelist))
                (setq pecntr (addi pecntr))

                (cond ( (eq pecntr numofedges)(terpr)(return t)))

                (go pnloop)))


(defun seties ()

; For each node, count the number of incoming edges.
;

        (prog (cntr)

                (array ies fixnum numofnodes 1)
                (setq cntr numofnodes)
        loop
                (setq cntr (subi cntr))
                (store (ies cntr 0) (length (neighbor cntr 0)))
                (cond ( (zerop cntr) (return t)))
                (go loop)))
```



```
; ************************************************************
;                                                            ;
;       PROCEDURE KPART                                      ;
;                                                            ;
;               This procedure partitions the MEG optimally  ;
;               (minimum number of partitions), given the    ;
;               maximum allowed stage time together with     ;
;               Dss and Dsp.                                 ;
;                                                            ;
;       Sub-Procedures:                                      ;
;                                                            ;
;               1. checklmax    checks the input stage time  ;
;                               and returns error message    ;
;                               if it is not realizable.     ;
;                                                            ;
;               2. initgraph    initializes the MEG for the  ;
;                               stage partitioning.          ;
;                                                            ;
;               3. getcut       actually carries out the     ;
;                               stage partitioning.          ;
;                                                            ;
;               4. onepart      gets the next stage.         ;
;                                                            ;
;               5. printcutset  prints out the cutsets.      ;
;                                                            ;
;               6. printspd     prints out the resultant     ;
;                               stage times.                 ;
; ************************************************************


(defun rkpart (lmax dss dsp)

   (prog (k)

        ; if lmax is not realizable, return an error message.
        (cond ((not (checklmax)) (return nil)))

        (initgraph)
        (setq k (getcut))
        (printcutset)
        (printspd)
        (return k)
    )
)
```

```
(defun checklmax ()

;
;       Description: This function checks if the input
;               stage time is realizable
;

    (prog(minlmax)

        ; compute the minimum realizable stage time.
        (setq minlmax (fminlmax))

        (cond ( (not (greaterp minlmax lmax))
               (return t)))

        (terpr)
        (terpr)
        (cprintf '%10s 'ERROR:)
        (cprintf '%5s     'Lmax)
        (cprintf '%5d   lmax)
        (cprintf '%3s   'is)
        (cprintf '%4s   'too)
        (cprintf '%7s   'small.)
        (cprintf '%6s   'Must)
        (cprintf '%3s   'be)
        (cprintf '%7s   'longer)
        (cprintf '%5s   'than)
        (cprintf '%5d    minlmax)
        (cprintf '%7s   'nsec.)
        (terpr)
        (return nil)
    )
)


(defun initgraph ()

;
;       Description: Initialize all the matrices and buffers
;

    (prog ()

        ; declare global variables, cutset and spd.
        (array cutset t numofnodes 1)
        (array spd fixnum numofnodes 1)

        ; clear all buffer area.
        (fillarray 'nmat1 '(0))
        (fillarray 'nmat2 '(0))
        (fillarray 'nmat3 '(0))
        (return t)))
```



```
(defun getcut ()

;
;       Description: Stage partition the MEG. The variable
;               "termw" remembers the weights of the paths
;               terminated by any register.
;

    (prog (k termw)

        (setq eh '())
        (setq k 0)

        ; get all the root nodes as the starting nodes
        ; of the first stage

        (setq nh (zerocols 'adjm))
loopgc
        (setq termw '(0))

        ; initialize the weight of the stage-starting
        ; nodes.

        (initweight)

        (setq k (add1 k))
        (onepart)
        (store (cutset k 0) (append eh (cutset k 0)))


        ; clear the weights of the nodes next to the
        ; current stage latches, since not all these
        ; nodes are in "nh" and is to be initialized.

        (cwcut)
        (cond ((zerop (length nh))
               (store (spd k 0) (max (spd k 0) (apply 'max termw)))
               (return k)))

        ; if not the last stage, add dss to the
        ; stage time.

        (store (spd k 0) (add dss (spd k 0)))

        ; compare to the path weight terminated due to
        ; the registers and choose the lager one.

        (store (spd k 0) (max (spd k 0) (apply 'max termw)))
        (go loopgc)))
```

```
(defun initweight ()

;
;       Description: Initialize the weights of the nodes
;               which becomes the starting nodes of the
;               next stage. (For a root node, the weight
;               is its mpd. For any non-root node, the
;               weight is its mpd plus dsp.
;

        (prog (iwcntr addweight iwnid)

              (setq addweight 0)
              (cond ((not (zerop k))
                     (setq addweight (add addweight dsp))))
              (setq iwcntr (length nh))
        iwloop
              (setq iwnid (nthelem iwcntr nh))
              (store (nmat3 iwnid 0)
                     (plus addweight
                           (nth iwnid mpd)))
              (setq iwcntr (subi iwcntr))
              (cond ((zerop iwcntr) (return t)))
              (go iwloop))]


(defun cwcut ()

;
;       Description: Sets the weight of the nodes connected
;               to the current cut-line to zero before
;               calling the "initweight" function. (nmat3)
;

        (prog (cecntr cweid cwnid)

              (setq cecntr (length (cutset k 0)))
              (cond ((zerop cecntr) (return t)))
        cwloop
              (setq cweid (nthelem cecntr (cutset k 0)))
              (setq cwnid
                    (eval (caddr (nth cweid edgelist))))
              (store (nmat3 cwnid 0) 0)
              (setq cecntr (subi cecntr))
              (cond ((zerop cecntr) (return t)))
              (go cwloop)))
```



```
(defun onepart ()

;
;       Description: Get the next stage.
;

    (prog ()
          (setq temp '())
          (getreadynodes)
          (setq nh '())
loopo
          (onespan)
          (cond ((zerop (length temp)) (return t)))
          (go loopo)))


(defun getreadynodes ()

;
;       Description: Get all the nodes all the input edges
;               of which are already traversed. Put them
;               in "next head".
;

    (prog (rcntr rnid)
          (setq rcntr (length nh))
    looprdy
          (setq rnid (nthelem rcntr nh))
          (cond ((eq (nmat1 rnid 0) (ies rnid 0))
                 (setq temp (cons rnid temp))))
          (setq rcntr (subi rcntr))
          (cond ((zerop rcntr) (return t)))
          (go looprdy)))


(defun onespan ()

;
;       Description: Check the current searching fronts
;               (nodes) whether they can be included in
;               the current partition.
;

        (setq sf temp)
        (setq temp '())
        (setq se '())
        (part1)
        (part2)
        (part3)
)
```

```
(defun part1 ()

;
;       Description: Weight all the children of the searching
;                    fronts. Mark the visited edges. Also, increment
;                    the node-visit-count (nmat1).
;

    (prog (cntrsf sfnid cntrp1 eidp1 tonodep1)
          (setq cntrsf (length sf))
    p1oloop
          (setq sfnid (nthelem cntrsf sf))
          (store (spd k 0) (max (spd k 0) (nmat3 sfnid 0)))
          (setq cntrp1 (length (incdm sfnid 1)))
    p1iloop
          (cond ((zerop cntrp1) (go noedges)))
          (setq eidp1 (nthelem cntrp1 (incdm sfnid 1)))
          (setq eh (cons eidp1 eh))
          (setq tonodep1
                (eval (caddr (nth eidp1 edgelist))))
          (store (nmat1 tonodep1 0) (add1 (nmat1 tonodep1 0)))


          ; update the weight of the child node
          ; (of the current searching front node)
          (updateweight eidp1 sfnid tonodep1)
          (setq cntrp1 (sub1 cntrp1))
          (cond ( (not (zerop cntrp1)) (go p1iloop)))
    noedges
          (setq cntrsf (sub1 cntrsf))
          (cond ( (zerop cntrsf) (return t)))
          (go p1oloop)))



(defun updateweight (uwe ufrom uto)

  (prog (newweight)
    (if   (zerop (plus (cadddr (nth uwe edgelist))
                       (caddddr (nth uwe edgelist))))

     then (setq newweight
                (max (nmat3 uto 0)
                     (plus (nmat3 ufrom 0)
                           (nth uto mpd))))
     else (setq newweight
                (max (nmat3 uto 0)
                     (plus dsp (nth uto mpd))))
          (setq terms (cons (plus (nmat3 ufrom 0) dss) terms))
    )
    (store (nmat3 uto 0) newweight)
    (return t)))
```



```
(defun part2 ()

;
;       Description: Check all the children of the searching
;                    heads and get all the nodes all of whose
;                    input edges are marked.
;

    (prog (cntrsf sfnid cntrp2 oedge childp2)

          (setq cntrsf (length sf))
    p2oloop
          (setq sfnid (nthelem cntrsf sf))
          (setq cntrp2 (length (incdm sfnid 1)))
          (cond ((zerop cntrp2) (go nooedge)))
    p2iloop
          (setq oedge (nthelem cntrp2 (incdm sfnid 1)))
          (setq childp2
                (eval (caddr (nth oedge edgelist))))
          (cond ( (eq (nmat1 childp2 0) (ies childp2 0))
                  (getse childp2)))
          (setq cntrp2 (sub1 cntrp2))
          (cond ( (not (zerop cntrp2)) (go p2iloop)))
    nooedge
          (setq cntrsf (sub1 cntrsf))
          (cond ( (zerop cntrsf) (return t)))
          (go p2oloop)))


(defun getse (senodeid)

;
; if the node with all its incoming edges marked has not
; ben marked yet, then mark the node and put it in the
; new searching-end node sets.
;

    (cond ( (zerop (nmat2 senodeid 0))
            (setq se (cons senodeid se))
            (store (nmat2 senodeid 0) 1)
          )
    )
)
```

```
(defun part3 ()

;
;       Description: Check all the searching ends and
;               determine the nodes which can be included
;               in the current partition. Also update the
;               cutset and current maximum stage time.
;

     (prog (cntrp3 nidp3 dss0)

        (setq cntrp3 (length se))
        (cond ((zerop cntrp3) (return t)))
     p3loop
        (setq nidp3 (nthelem cntrp3 se))
        (setq cntrp3 (sub1 cntrp3))
        (setq eh (sublist eh (incdm nidp3 0)))

        ; If nidp3 is a terminal node then do not add dss
        (setq dss0 dss)
        (cond ((zerop (rowsum 'adjm nidp3)) (setq dss0 0)))

        ; check if node nidp3 can be added to the current partition
        (if (greaterp (plus dss0 (nmat3 nidp3 0)) lmax)
             then
                (setq nh (cons nidp3 nh))
                (store (cutset k 0)
                       (append (cutset k 0) (incdm nidp3 0)))
                (store (nmat3 nidp3 0)
                       (plus (nth nidp3 mpd) dsp))
             else
                (store (spd k 0) (max (spd k 0) (nmat3  nidp3 0)))
                (if (not (zerop (rowsum 'adjm nidp3)))
                    then (setq temp (cons nidp3 temp))
        ))
        (cond ((zerop cntrp3) (return t)))
        (go p3loop)))


(defun sublist (from to)

;
;       Description: Returns the difference (from - to)
;               of the lists from and to.
;

        (prog (cntrsub)
                (setq cntrsub (length to))
        loop
                (setq from (delete (nthelem cntrsub to) from))
                (setq cntrsub (sub1 cntrsub))
                (cond ( (zerop cntrsub) (return from)))
                (go loop)))
```



```
(defun printcutset ()

        (prog (pcscntr csl)

                (cond ( (null (cutset 1 0))
                        (terpr) (terpr)
                        (print "No Cutsets.")
                        (print '(K = 1))
                        (return t)))

                (setq pcscntr 1)
                (terpr)
                (cprintf '%12s '**********)
                (cprintf '%14s 'EDGE-CUTSETs)
                (cprintf '%12s '**********)
                (terpr)
        pcsloop
                (terpr)
                (setq csl (cutset pcscntr 0))
                (cond ( (zerop (length csl)) (return t)))
                (cprintf '%8s 'cutset)
                (cprintf '%3d pcscntr)
                (cprintf '%3s '=)
                (print csl)
                (setq pcscntr (add1 pcscntr))
                (go pcsloop)))


(defun printspd ()

        (prog (pspdcntr spdnum)
                (setq pspdcntr 1)
                (terpr)
                (terpr)
                (cprintf '%6s '*****)
                (cprintf '%25s 'STAGE-PROPAGATION-DELAY)
                (cprintf '%7s '*****)
                (terpr)
                (terpr)
    pspdloop
                (setq spdnum (spd pspdcntr 0))
                (cond ( (zerop spdnum) (return t)))
                (cprintf '%15s 'stage-delay)
                (cprintf '%3d pspdcntr)
                (cprintf '%3s '=)
                (print (spd pspdcntr 0))
                (terpr)
                (setq pspdcntr (add1 pspdcntr))
                (go pspdloop)))
```

```
;*************************************************************
;
;       PROCEDURE OPART
;
;               This procedure finds an optimal stage
;               partitioning (minimal clock period) when
;               the number of stages is fixed.
;
;               This routine calls the KPART procedure with
;               the stage times chosen in a binary-search-
;               like fashion out of all the possible stage
;               times computed by the procedure, ienum.
;               The search is stopped when no shorter stage
;               time which results the desired number of
;               stages can be found.
;
;       Sub-Procedure
;
;               1. ienum        computes all the realizable
;                               stage times.
;*************************************************************


(defun opart (numofptn)

    (prog (ub lb crtptr oldptr k)

        (cond ((lessp numofptn 1)
        (setq numofptn 1)
        (print "Warning: Number of Partitions can not be less than 1")))

        ; compute all the realizable stage times
        (setq intervals (ienum))

        ; print out the intervals

        (terpr) (terpr)
        (print "Realizable Stage Times are:")
        (terpr) (terpr)
        (print intervals)
        (terpr) (terpr)

        (setq ub (length intervals))
        (setq crtptr (/ (add1 ub) 2))
        (setq oldptr (/ (add1 ub) 2))
        (setq lb 0)
    loop
        (terpr)
        (print "Enter rkpart with stage time:")
```



```
        (cprintf '%4d (nthelem crtptr intervals))
        (cprintf '%6s '--->)

        (setq k (rkpartnp (nthelem crtptr intervals) dss dsp))

        (cprintf '%3s 'k) (cprintf '%2s '=) (cprintf '%3d k)
        (terpr)

        (if     (not (greaterp k numofptn))
         then   (setq ub crtptr) (setq oldptr crtptr)
                (setq crtptr (diff crtptr (/ (add1 (diff crtptr lb)) 2)))
                (cond ((zerop crtptr)(setq crtptr 1)(go exit)))
                (go loop))

        (if     (greaterp k numofptn)
         then   (setq lb crtptr)
                (setq crtptr (plus crtptr (/ (add1 (diff ub crtptr)) 2))))
        (cond ((eq crtptr 1)(go exit)))
        (cond ((neq crtptr oldptr)(go loop)))
    exit
        (terpr)
        (terpr)
        (print "Enter rkpart with stage time:")
        (cprintf '%4d (nthelem crtptr intervals))
        (terpr)
        (rkpart (nthelem crtptr intervals) dss dsp)
        (terpr)
        (cprintf '%5s 'dmax)
        (cprintf '%1s '=)
        (cprintf '%4d (nthelem crtptr intervals))
        (terpr)
        (terpr)
        (return t)))



(defun rkpartnp (lmax dss dsp)

;
; This function is identical to the procedure, ikpart,
; except that the printing statements are removed.
;

   (prog (k)
        (initgraph)
        (setq k (getcut))
        (return k)
    )
)
```

```
;*************************************************************
;
;       PROCEDURE CP
;
;               This procedure computes several different
;               types of critical path in a MEG (or in an
;               acyclic directed graph).
;               The three important types of critical paths
;               are that of control flow, data flow and
;               clock cycle.
;
;       Sub-procedures
;
;               1. glabel       is described in "fpool".
;
;               2. cpdf         computes data flow CP.
;
;               3. other critical path finding procedures
;                  are to be added.
;*************************************************************


(defun cp (dss dsp)

    (array cpmarkn fixnum numofnodes 4)
    (array cpmarke fixnum numofedges 4)
    (setq df 0)
    (setq cf 1)
    (setq clk 2)
    (setq mi 3)
    (glabel)
    (cpdf)
)


(defun cpdf ()

;       Description: This routine computes the critical
;               path(s) of a MEG for the data flow of the
;               associated micro cycle. The data-flow
;               critical path delay(s) from the root node(s)
;               to each node are computed and put in nmati
;               matrix. Also, the nodes and edges on the
;               critical path(s) are marked.
;
```



```
        (weigh_df)
        (mark_df)
        (terpr)
        (print 'CRITICAL_PATH_NODES)
        (terpr)
        (terpr)
        (apr3 'cpmarkn)
        (terpr)
        (terpr)
        (print 'CRITICAL_PATH_EDGES)
        (terpr)
        (terpr)
        (apr3 'cpmarke)
        (terpr)
)


(defun weigh_df ()
        (prog (dset cntr0)
                (fillarray 'nmati '(0))
                (mapcar 'iwcpdf (zerocols 'adjm))
                (setq cntr0 0)
        loop
                (setq dset (depthmat cntr0 0))
                (cond ((null dset) (return t)))
                (mapcar 'wfis dset)
                (setq cntr0 (add1 cntr0))
                (cond ((eq cntr0 numofnodes)(return t)))
        (go loop)))


(defun iwcpdf (nodeid)

        (store (nmati nodeid 0) (nth  nodeid mpd)))


(defun wfis (fnode)
        (prog (oedges)
                (setq oedges (incdm fnode 1))
                (cond ((null oedges)(return)))
                (mapcar 'wfin oedges)))
```

```
(defun wf1n (oedge1d)
  (prog (oedge father cnode numreg)
    (setq oedge (nth oedge1d edgelist))
    (setq father (eval(cadr oedge)))
    (setq cnode (eval(caddr oedge)))
    (setq numreg (add (caddddr oedge) (cadddddr oedge)))
    (store (nmat1 cnode 0)
      (max (nmat1 cnode 0)
           (plus  (nmat1 father 0)
                  (nth cnode mpd)
                  (times (plus dss dsp) numreg)
)))))


(defun mark_df ()

    (prog (temp lbuf max weight cntr cntr1 node)
        (setq temp (zerorows 'adjm))
        (setq lbuf '())
        (setq max 0)

        ; get all the terminal nodes the cpweight of which
        ; is maximum.

        (setq  cntr (length temp))
    loop0
        (setq node (nthelem cntr temp))
        (setq weight (nmat1 node 0))
        (if (greaterp weight max) then (setq max weight)
                                       (setq lbuf (list node))
         else (if (eq weight max) then (setq lbuf (cons node lbuf))))
        (setq cntr (sub1 cntr))
        (cond ((not (zerop cntr)) (go loop0)))

    loop1
        (mapcar '(lambda (node) (store (cpmarkn node df) 1)) lbuf)
        (setq temp '())
        (setq cntr (length lbuf))
    loop2
        (setq node (nthelem cntr lbuf))
        (setq temp (append temp (echeck node)))
        (setq cntr (sub1 cntr))
        (cond ((not (zerop cntr)) (go loop2)))

        (cond ((null temp) (return t)))
        (setq lbuf temp)
        (go loop1)
    ))
```



```
(defun echeck (ecnode)
        (prog (cntrec edge orgnode numreg ntmp)
        (setq inedges (incdm ecnode 0))
        (cond ((null inedges)(return nil)))
        (setq ntmp '())
        (setq cntrec (length inedges))
        loope

        (setq edge (nth (nthelem cntrec inedges) edgelist))
        (setq orgnode (eval (cadr edge)))
        (setq numreg (plus (caddddr edge) (cadddddr edge)))
        (if (eq (plus (nmat1 orgnode 0)
                      (times (plus dss dsp) numreg)
                       (nth ecnode mpd)
                )
                (nmat1 node 0)
            )
         then (store (cpmarke (nthelem cntrec inedges) df) 1)
              (setq ntmp (cons orgnode ntmp))
        )
        (setq cntrec (sub1 cntrec))
        (cond ((zerop cntrec) (return ntmp)))
        (go loope)
))
```

```
;*************************************************************;
;                                                             ;
;       FUNCTION INDEX (IN ALPHABETICAL ORDER)                ;
;                                                             ;
;*************************************************************;
```

```
fminlmax ()
gen_depth_height ()
getcut ()
getreadynodes ()
getse (senodeid)
glabel ()
ienum ()
ifind ()
init_buffers ()
initgraph ()
initweight ()
inputprocess (nodeinput edgeinput)
iopt (row1)
iwcpdf (nodeid)
mark_df ()
name2id (namelist)
nflabel (sset)
ngdepth ()
nodeprocess (nodelistfilename)
onepart ()
onespan ()
opart (numofptn)
part1 ()
part2 ()
part3 ()
pelist ()
printcutset ()
printnodelist ()
printspd ()
rkpart (lmax dss dsp)
rkpartnp (lmax dss dsp)
rowsum (name iy)
seties ()
sublist (from to)
updateweight (uwe ufrom uto)
weigh_df ()
wfin (oedgeid)
wfis (fnode)
zerocols (name)
zerorows (name)
```