On Automatic Verification of SLIDE Descriptions

by

John J. Wallace

DRC-01-2-80

August 1979

On Automatic Verification of SLIDE Descriptions

John J. Wallace

Bell Laboratories
Warrenville-Naperville Rd.
Naperville, Il. 60540

Dept. of Electrical Engineering
Carnegie-Mellon University
Pittsburgh, Pa. 15213

August 1979

Submitted to the Electrical Engineering Department of Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Master of Science.

Project committee:

- Alice Parker
- Mario Barbacci
- Nico Habermann

This research has been supported by the U. S. Army Research Office under grants *DAAG29-76-G-0224 and »DAAG29-78-G-0070. Support for the author was provided by Bell Laboratories.

Table of Contents

Abstract

Due to the rise in both the use of hardware descriptive languages and the interest in communications link design, the need to verify descriptions of communication links has become an important step in the design cycle. This paper discusses some of the issues involved in automatically verifying hardware descriptions written in SLIDE, an I/O hardware descriptive language.

We describe a communications link in terms of its synchronization mechanism, which contains synchronization hardware and a synchronization protocol. A syntax is introduced which is amenable to describing synchronization mechanisms and to automatic verification of the mechanisms. A synchronization capsule is used to describe a synchronization mechanism; processes and roles are used to describe the synchronization hardware and synchronization protocol respectively. A path expression is used to specify the design specs.

A formalism is introduced which can be used to automatically verify the synchronization mechanism described by a synchronization capsule. This formalism is based upon Petri Nets and Vector Addition Systems.

This paper was created with the aid of the text processing program SCRIBE and the graphics programs SPACS and SPXIMG. The files for this paper reside in two subdirectory master files, VER.MAS and VERPIC.MAS on account [X375JW76] on CMU-10D.

Acknowledgements

I would like to extend thanks and gratitude to the many people **who** assisted **with this** work, especially my committee members: Mario Barbacci provided input **for the SLIDE and** V-SLIDE languages; Nico Habermann suggested **the** concept **of a role; and Alice Parker** provided direction and filled in the gaps in my knowledge^bf hardware.

Sten Andlcr provided valuable input concerning the use **of path expressions. Art Altman** has been involved with SLIDE from the start and provided advice on semantic issues. **Vittal** Kini helped develop the Petri Net and Vector Addition System formalisms.

I would also like to thank the document production and graphics people in **the CMU** Computer Science Department. Their excellent software packages **were a great aid in the** preparation **of** this paper.

1. Introduction

This paper describes the work done over the past year on verification issues in SLIDE, an I/O hardware descriptive language. At least a cursory Knowledge of SLIDE [Parker 78, Wallace 79] and the aims of the CMU-DA group at Carnegie-Mellon University [Parker 79a] is assumed. Someone without this knowledge may still be interested in Chapters 3, 4, 5, and 7, and should definitely read Chapter 2 on terminology.

1.1 The Importance of Hardware Descriptive Languages

With the advent of LSI and VLSI technology, the need for digital hardware design techniques which differ from the ad hoc techniques used previously has risen. This is because of the complexity of the circuits which can now be implemented in VLSI. Hardware descriptive languages (HDLs), such as ISPS [Barbacci 78], are an attempt to formalize the design procedure. An entire central processor (processor, memory, etc.) can be modeled by an ISPS description. This description can then serve as a formal basis for documentation, simulation, verification, and computer aided design of the actual hardware.

For a complete discussion of HDLs see [Barbacci 75] and [CHDL 75].

1.2 The Importance of Communications Links

Another result of VLSI technology has been the increased availability of inexpensive processors and large memories, two components which have traditionally been the most expensive components of a computer. Consequently, it has become economically feasible to combine many processors and memories to form a multi-computer network, or to use one or more processors and memories as components in a larger system.

As a result of this, a third component, the communications link, is beginning to receive its share of attention [Chen 74, Pagel]. This can be attributed to the following reasons:

1. The cost of the communications link has risen relative to the costs of processors and memories.

2. As processor and memory speed increases, the communications link is becoming the bottleneck in computers.

3. Because components connected to the communications link are becoming more intelligent, more complex links are possible (and more complex links are needed).

For the purposes of this paper, a communications link is anything that interconnects processors, memories, peripherals, or other communication links. This includes buses, crosspoint switches, interfaces, etc.

Because of the increased interest in communications link design, much research is beginning to be conducted on this topic [Chen 74, Ansi 79, Levy 78, Burr 79]. Part of this research at Carnegie-Mellon University has resulted in an I/O hardware descriptive language (IOHDL) designed specifically for writing descriptions of communication links. The language, called SLIDE for *Structured Language for Interface Description and Evaluation*, is amenable to writing descriptions of peripherals, controllers, interfaces, buses, and other communications related hardware [Parker 78, Wallace 79].

1.3 The Need for Verification

We need confidence in the systems we design; i.e. we need verifiably correct hardware. Consequently, we want to write verifiably correct hardware descriptions. From these we can produce correct hardware, assuming our computer aided design procedures preserve correctness.

The need to verify correct operation is stronger when designing communication links. This is because most communication links have a synchronization mechanism which synchronizes the actions of various independent processes connected to the link. This synchronization mechanism usually includes a piece of hardware which synchronizes the actions of the various processes, and a protocol which each of the processes must obey for the communications link to operate correctly. For example, a bus usually has a bus arbiter which arbitrates between pending bus requests, and a protocol which all processes interfaced to the bus must obey.

The design of a synchronizing mechanism tends to be error prone. This is because:

1. The mechanism can be complex.

2. Loci of control can reside in many independent processes.

When verifying SLIDE descriptions, we want to verify the correct operation of the synchronizing mechanism, and make sure that all processes utilizing the mechanism obey the protocol. Although this work was conducted with SLIDE in mind, the results can be applied to any HDL which allows interaction between independent processes.

The use of HDLs represents a step which blurs the distinction between hardware design and software design. Consequently, there are many parallels between the efforts to verify hardware descriptions and those to verify software. Specifically, concepts such as abstraction and encapsulation (introduced later in this paper) have their roots in software language design and software verification.

1.4 Methodology for Automatic Verification of SLIDE Descriptions

This paper provides a framework for a SLIDE verification system. As mentioned previously, we want to verify the synchronization between independent processes. We call this global verification and distinguish it from local verification which is determining whether an individual process performs its intended function. Although local verification is an important issue, we will not be concerned with it here.

SLIDE provides a good starting point for a language which is amenable to verification. It is behavioral; that is it describes the behavior of hardware without describing an implementation. Also, actions are described explicitely in SLIDE, at a high (register transfer like) level as opposed to a low (gate) level.

We take the attitude that we want to write verifiably correct SLIDE descriptions; we should have verification in mind when we design hardware. Also, we believe it is instructive for the designer to structure his design using the constructs of abstraction and encapsulation (described later) to aid in the design and verification procedure. The SLIDE syntax should provide adequate facilities to do these things. This is a strong parallel to the software structured programming design methodology.

We will describe a modified SLIDE syntax (called V-SLIDE for "Verifiable SLIDE") which provides these facilities. We will also outline some techniques which can be automated that will do global verification on a V-SLIDE description.

The feasibility and limitations of these techniques will be discussed as they are introduced.

1.5 Outline of Paper

The goal of this paper is to explore the many different issues involved in automatically verifying SLIDE descriptions. The paper is filled with ideas which are crying to be implemented. Unfortunately, there was no time to both explore all of the verification issues involved and implement the findings. In that respect, this work is incomplete. However, techniques are outlined, and possible implementations are sometimes suggested.

Chapter 2 discusses some basic terminology. Chapter 3 discusses our views on verification. Chapter 4 introduces the V-SLIDE syntax. Chapters 5 and 6 introduce Petri Nets and Vector Addition Systems, the formalisms we will be using, and Chapter 7 discusses the techniques for automatic verification.

2. Terminology

This chapter describes some terms which are basic to this paper. Other terminology is introduced when needed.

Process: An independent executing environment, a piece of hardware. In SLIDE, a process is the part of a SLIDE description which describes an independent executing environment. The difference between the description and the hardware described is only one of representation.

Processes can interact with one and another in various ways.

We draw a constant analogy between processes and Petri Nets (Chapter 5).

Execution, actions: A process executes. In doing so, it performs zero or more time ordered actions which are described by SLIDE statements. In a Petri Net, actions are represented by transitions, and execution is represented by the firing of transitions.

Blocked: The execution of a process may be blocked if one of its actions (such as a delay statement) can not (yet) execute completely (occur). In a Petri Net, this is represented by a transition which is partially enabled. Blocking may or may not be temporary.

3. Verification of Hardware Descriptive Languages

This chapter is an introduction to our views on verification, the types of verification we are concerned with, and our solutions.

3.1 Reasons for Verification

We see automatic verification as a necessary and natural step in the design cycle. Verification, combined with adequate simulation and testing, gives us confidence in the system we are creating.

Verification is important now that complex hardware is being designed and implemented with the aid of HDLs. This is especially true when designing communications links where many independent processes interact.

3.1.1 SLIDE Design Cycle

Figure 3-1 shows the SLIDE design cycle. The initial design is formulated into a SLIDE description which includes verification specifications. This description is run through the automatic verifier which determines if the description is "correct", and checks for various design flaws. The results of verification may lead to subsequent redesign and reverification. The SLIDE description can then be compiled and simulated. Results here may lead to more redesign, a different parameterization, or a different design altogether. Finally, when the simulation of the SLIDE description operates as intended, the circuits for the hardware will be automatically built.[1]

3.2 Types of Verification

We are concerned with the verification of the synchronization of independent processes. This is called global verification. The types of things we would like to verify are:

- Error recovery: The ability to recover from faults.
- Reliability: Freedom from the occurence of faults.
- Specs: There are certain invariants which describe how hardware should operate. These are called its "specifications" or "specs". We want to verify that a SLIDE description adheres to its specs.

[1] The verifier could use either the original description or the compiled GDB for input. In fact, using the GDB would be simpler since the GDB has a regular syntax The current compiler, however, takes standard SLIDE as input, whereas fhe verifier will take a modified SLIDE, called V-SLIDE, as input. Instead of changing the current compiler, it might be simpler in the short run to create a preprocessor which translates V-SLIDE into SLIDE. This way, a designer can write one description in V-SLIDE and then use the verifier, preprocessor and compiler, simulator, and other software packages.

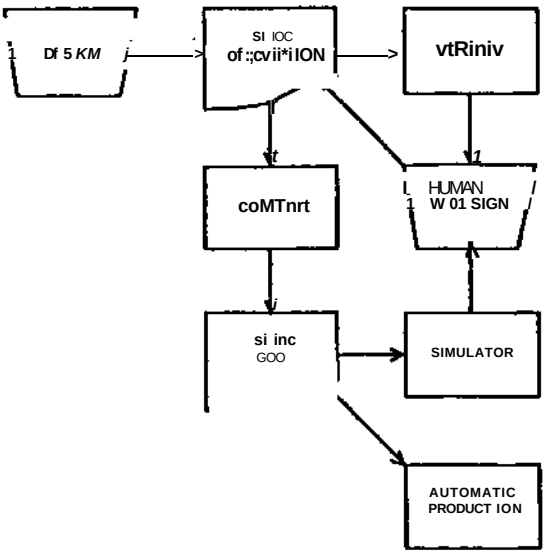


Figure 3-1: SLIDE Design Cycle

For example, the specs for a bus would indicate that no two processes can be using the bus concurrently, that a certain priority relationship exists between processes, that a certain protocol must be followed by all processes using the bus, etc. We would want to verify that the bus hardware and all processes using the bus adhered to these specs.

- Unnecessary constraints on concurrency: Although a SLIDE description should adhere to its specs, it should allow as much concurrency as possible. This gives maximum throughput.

  Unnecessary constraints on concurrency don't always indicate design errors. Doing something sequentially rather than concurrently may be an economic decision. For example, the specs for a bus may indicate that the next process to use the bus may be selected while the current process is using the bus. The designer, however, may opt for sequential operation. The verifier should point out to the designer any unnecessary constraints on concurrency (including this one).

- Deadlock: If a process is blocked pending arrival of a control signal which will never arrive, the process is cieadlocked. This may happen if several processes are cyclically waiting for signals from one another, or if a signal is lost [Huen 73]. This is similar to deadlock in the classical software sense [Habermann 69].*

- Starvation: If execution of one or more processes effectively blocks execution of another, the blocked process is starved, and the system is in a starvation state.

---

[1] Huen discusses another type of deadlock which may arise when a SLIDE description is implemented in hardware [Huen 73, ChaptV). This deadlock arises due to errors in the implementation process, not in the actual design. We will not be concerned with this.

Starvation is analogous to deadlock since a process is blocked. Its occurence, however, depends on the dynamic behavior of the entire system.

The possibility of starvation in a system is common. For example, if saturated with high priority interrupts, an interrupt mechanism would not be able to honor any low priority interrupts. Any possibilities of starvation should be pointed **out** to the designer.

- Reception: If a process receives a control signal when it is not in a state to honor it, and consequently, the signal is lost, a reception error has occured. This may happen if two communicating processes become unsynchronized [West 78].

- Overflow: There are two types of overflow errors: overflow of lines (when information is output over a line too fast, and interference occurs) and overflow of buffers (when buffers are written into faster than they are read from). We want to verify that no errors can occur due to overflow.

3.3 Our Solutions

The rest of this paper discusses our methods for automatically verifying the above. We will propose a syntax that is built upon SLIDE and is amenable to verification. Using this syntax, a designer can specify both the hardware specs and the hardware description.

We will not be concerned with error recovery or reliability. This is because these are areas of their own, and a lot of research is currently being conducted here (for example see [IEEE 78]).

For the other verification issues, we will propose a formalism which can be used **to** aid in verification.

4. .V-SLIDE Syntax

This chapter introduces the V-SLIDE ("Verifiable SLIDE") syntax. We discuss what motivated us to develop this syntax and the syntax itself. A simple example, a handshake, and then a more complex example based on the UNIBUS* are given. We then discuss the problem of interconnecting V-SLIDE descriptions in various configurations.

4.1 Motivation

We need a syntax which is conducive to writing verifiable descriptions and to automatic verification. By "writing verifiable descriptions" we mean that the designer should (1) design with verification in mind, and (2) structure his design in such a way as to minimize the possibility of a design error.

The V-SLIDE syntax should support these things. Specifically, it should provide facilities for [Ichbiah 79, Page4-1]:

1. Encapsulation: Knowledge about common actions should be collected in one place, called a capsule. Things defined within a capsule may be local and are not visible outride the capsule; others may be exported and are visible outside the capsule. Encapsulation enhances maintainability since a change in the actions has only to be effected at a single place.

2. Abstraction: There should exist a clear distinction between the abstract name of an action and its definition. Definitions of common actions should be encapsulated together; only the names of the actions are exported. A process can then use the abstract action name and not be concerned with its definition.

3- A total design system: Any system may be composed of many pieces of hardware which are described by many V-SLIDE descriptions residing in separate files. Actions used in one description (file) will be defined in another. This concept of definition and instantiation goes hand in hand with the concepts of encapsulation and abstraction.

4. Formally specifying the design specs: We need a formal syntax for specifying the design specs to the verifier. These formal design specs should be an integral part of the hardware description, but they should be separate from the actual design. They are redundant in that they should be implied by the design.

---

*UNIBUS is a registered trademark of Digital Equipment Corporation.

4.2 Proposed Syntax

In this section, we discuss the proposed V-SLIDE syntax which is essentially an extension to SLIDE. We focus on the concepts involved — not on trying to develop a rigid syntax.

4.2.1 The Module

The module is to V-SLIDE what the main process is to SLIDE. Each V-SLIDE description (file) is exactly one module.

A module is a capsule. Within a module, there are hardware declarations (registers, lines, etc.), process declarations, and synchronization capsule declarations (discussed later). Any hardware declared immediately within a module (i.e. not within one of the processes or synchronization capsules) is considered global. These registers, lines, etc. form the "hooks" that allow many modules to be interconnected.

The concept of a module was introduced for two reasons:

1. It keeps the concept of a process consistent. A process describes a piece of hardware; a module is a capsule.

2. Since a module contains no statements, it eliminates the need for the dummy loop statement which is always included in a SLIDE main process.

4.2.2 Tho Process

The process in V-SLIDE is exactly the same as a process in SLIDE. It describes an independent piece of hardware and may contain hardware declarations, subprocesses, and statements.

4.2.3 The Synchronization Capsule

A synchronization mechanism synchronizes the actions of the independent processes which use the mechanism (see Section 1.3). It usually consists of two closely related parts:

1. Hardware which synchronizes Ihe actions of the processes. This is called the synchronization hardware.

2. A protocol which must be followed by all processes using the mechanism for it to operate correctly. This is called the synchronization protocol.

For example, a bus usually consists of an arbiter which synchronizes the actions of processes which access the bus (this is the synchronization hardware) and a protocol which these

processes must adhere to (this is the synchronization protocol).

A synchronization capsule describes a synchronization mechanism. It consists of a header, a path expression, hardware declarations, processes, and roles. Before discussing these, we will introduce the concept of an abstract action.

4.2.3.1 Abstract Actions

An abstract action has two parts: a name and a definition. The definition is given only once, and the name refers to the definition.

There are three types of abstract actions:

1. An assignment abstract action has a definition which consists of one or more parallel assignments.

2. A delay abstract action has a definition which consists of a single SLIDE delay statement (with or without a timeout).

3. A conditional abstract action has a definition which consists of a V-SLIDE condition which evaluates to true (1) or false (0). It is usually used in an IF statement to control the performance of other abstract actions.

An abstract action occurs when the statements which define it complete. For an assignment abstract action, this is immediate. For a delay abstract action, this happens whenever the condition which the delay statement is delaying on becomes true. Conditional abstract actions either occur immediately if the condition is true, or do not occur at all if the condition is false. Abstract actions are performed by actors.

A word about notation. We use "a.b" to refer to action "b" performed by actor "a". Similarly, we use "a1.a2.b" to refer to action $^{M}b^{H}$ performed by an actor "a1" who is also actor "a2^{W}". We use "«*(a.b)^{W} to refer to the number of occurences of "a.b". The V-SLIDE syntax uses this notation except where the actor is implied by context in which case the "a." is omitted.

4.2.3.2 Header

We will now discuss the parts of a synchronization capsule.

The header states the name of the capsule and the names of the roles defined within it which will be exported (roles are discussed later). Figure 4-1 shows a sample header for a capsule by the name of "shake" with two exported roles defined within it. The roles* names are "shaker 1" and "shaker2^{M}. Note that in all V-SLIDE examples, reserved words are in upper

case; identifiers are in mixed lower and upper case.

```
SYNCHRONIZATION CAPSULE shake (ROLES shaker1, shaker2);
```

Figure 4-1: Synchronization Capsule Header

4.2.3.3 Path Expression

The path expression (PE) is used for verification purposes only. It indicates the design specs to the verifier by specifying which sequences of abstract action occurences are allowable.

We use the PE syntax defined in [Andler 79a] with a few modifications. Specifically, we use "NEXT" not ";" to indicate sequential operation, and we use ";" not "," to indicate parallel operation. This is consistent with SLIDE notation.

Also, any appearance of an abstract action name in a predicate (see [Andler 79a]) refers to the number of times that abstract action has occured.

Figure 4-2 shows a sample PE. Each abstract action name consists of two parts separated by a dot ("."). The first part indicates which process or role is to perform the action (e.g. "shaker1" or "shaker2"). We call this the actor. The second part indicates the action's name (e.g. "shake", "waitshake", etc.).

```
PATH
        (shaker1.shake   NEXT
         shaker2.waitshake  NEXT
         shaker2.ack   NEXT
         shaker1.waitack   NEXT
         shaker1.unshake   NEXT
         shaker2.waitunshake  NEXT
         shaker2.unack)*
END;
```

Figure 4-2: Path Expression

It is important to realize that the PE is a formal way of indicating the design specs to the verifier; it is not part of the design (i.e. it can be ignored by the simulator). In this example, the PE indicates that the action "waitshake" performed by actor "shaker2" must occur after the action "shake" performed by actor "shaker1" but before action "ack" performed by actor "shaker2". After the action "unack" performed by actor "shaker2" occurs, the whole

sequence may repeat.

Also, there is a significant but subtle difference between the use of "NEXT" and V in PEs and in processes or roles. In the former, they order the occurence of actions; in the later, they order the execution of statements.

From now on we use the notation discussed in Section 4.2.3.1.

Figure 4-3 shows a sample PE which uses predicates. Here, "requestor.req" can occur many times. Concurrently, "granlor.granta" can occur if the number of pending requests is between 1 and 3 inclusive, or "grantor.grantb" can occur if the number is greater than 3.

The "LET ... IN" construct defines a macro which is local to the PE.

Using the interpretation that the action "req" represents a request by the requestor, and the actions "granta" and "grantb" represent different types of grants from the grantor, this PE indicates that the requestor can make many requests, and the grantor concurrently makes two different types of grants depending on the number of pending requests.

```
PATH
    ! pr is the number of pending requests !
    LET
     pr = //(requestor,rcq) - //(grantor,granta) - //(grantor,grantb)
    IN
          requestor,req*:
          ( grantor,granta [pr>0 AND pr<4]
          + grantor,grantb [pr>3] )*
END:
```

Figure 4-3: Path Expression With Predicates

Different actors can perform distinct actions with similar names or they can perform the same actions. We differentiate between these cases as follows. First, if a similar action name is used by more than one actor, such as "Requestorl.Req" and "Requestor2.Req", then these are distinct actions, and each has its own distinct definition. The names are considered overloaded. The occurence of one does not effect the occurence of the other except where indicated in the PE. Overloading can be used to advantage as will be shown in a later example.

Second, if the same actor action pair is used more than once in the PE, then these refer to one action with one definition and one common occurence.

Third, if an action such as "Busllser.GrabBus" is qualified by another actor as in -Requestorl.BusUser.GrabBus" or "Requestor2.BusUser.GrabBus" (meaning action "GrabBus" performed by actor "BusUser" who is also actor "Requestor 1" or "Requestor2" respectively), then these refer to different occurences of the same action* (This is done in the memory bus description discussed at the end of this chapter.)

4.2.3.4 Hardware Declarations

Any hardware declarations (registers, lines, etc.) within a synchronization capsule are considered local to that capsule. The hardware declared may be accessed by processes and roles defined within the capsule, but not from outside the capsule.

4.2.3.5 Processes

Processes within synchronization capsules serve two purposes. They define the synchronization hardware, and they give the definitions of the abstract actions performed by the processes and used in the PE.

Abstract action definitions are given in parenthesis after the abstract actions[9] names. Figure 4-4 shows how abstract actions are defined in a process or role. Line 1 defines the conditional abstract action "CheckInlerrupt". Lines 3 and 5 define the assignment abstract actions "Ack" and "Unack". Line 4 defines the delay abstract action "WaitDropInterrupt". Only abstract action names are exported from the capsule, not the definitions.

```
[1] IF CheckInterrupt (intr EQL 1) THEN
[2]     BEGIN
[3]       Ack (sync «- /):
[4]           WaitDropInterrupt (DELAY UNTIL intr EQL \) NEXT
[5]       Unack (sync <- \)
[6]     END NEXT
```

Figure 4-4: Portion of Process or Role

4.2.3.6 Roles

Roles are used only within synchronization capsules. They define the synchronization protocol by defining a template which processes assuming the roles must adhere to. A process assumes a role by indicating this in its own process definition. Many processes can assume the same role.

Each role also gives the definitions of abstract actions used in the PE and performed by

processes assuming that role. Abstract action definitions are given in the same manner as discussed for processes (see Figure 4-4).

4.3 A Simple Example - A Handshake

Figure 4-5 shows the description of a synchronization mechanism for a bidirectional handshake. Figure 4-6 and 4-7 show the descriptions of two processes which handshake using this mechanism.

The handshake works as follows: the process assuming the role of "shaker1" (e.g. "controller") starts the handshake by raising the "11" line. Then the process assuming the role of "shaker2" (e.g. "interface") acknowledges by raising the "12" line. "Shaker1" then drops the "11" line, and "shaker2" drops the "12" line.

```
MODULE controllerModule:

    PROCESS controller (ROLES: handshake.shaker1);
            .
            .
        shake;  waitack  NEXT
        unshake
            .
            .
    END;
END;
```

Figure 4-6: Process Assuming Role of Shaker1

There are a few things about this example worth noting:

- The synchronization mechanism described is merely a protocol between two processes. Consequently for this example, there is no synchronization hardware and no processes defined within the synchronization capsule. The roles "shaker1" and "shaker2" define the synchronization protocol.

- Definitions for the abstract actions used in the PE are given in the roles. These definition are encapsulated within the roles. Only the names of abstract actions and the names of roles *are* exported from a synchronization capsule.

- Process "controller" states that it is to assume role "shakerl" by indicating this in its process header. When *an* abstract action name is used within this process, this is called an instantiation. It is like a macro call. The definition of the abstract action should be substituted for the name of the action by the preprocessor or compiler. The order of instantiations should match the template

```
MODULE handshake:

   SYNCHRONIZATION CAPSULE shake (ROLES shakerl, shaker2)t

        PATH
           (shakerl.shake  NEXT
            shaker2.waitshake  NEXT
            shaker2.ack  NEXT
            shakerl.waitack  NEXT
            shakerl.unshake  NEXT
            shakcr2.waitunshake  NEXT
            shaker2.unack)*
        END:

        OCAL LINE ll<>, I2<>;

        ROLE shakerl;
          OEGIN
            shake (II <- /)}  waitack (DELAY UNTIL 12 EQL /) NEXT
            unshake (II •- \)
          END:

        ROLE shaker2:
          BEGIN
            waitshake (DELAY UNTIL II EQL /) NEXT
            ack (12 •- / ) : wai tunshake (DELAY UNTIL II EQL \) NEXT
            unack (12 •- \)
          END;

   END Jof synchronization capsule shake!;
END lof module handshake!:
```

Figure 4-5: Handshake Module

```
MODULE intcrfaccModulc;

    •
  PROCESS Interface (ROLE handshake.shaker2):
        •
        •
      waitshake NEXT
      ack; uaitunshake NEXT
      unack
        •
        •
  END:
END;
```

Figure 4-7: Process Assuming Role of Shaker2

defined by the corresponding role in module "handshake".

- Process "controller" indicates in its process header that the role "shakerl* is defined in module "handshake". By convention, module "handshake" should be described in the file "HANDSHSLI". This allows the definition of abstract actions to be automatically accessed during preprocessing or compilation.

4.4 A Complex Example - A Memory Bus

The V-SLIDE description of a memory bus, called MemBus, which is based on the UNIBUS is contained in Appendix I. The system operates as follows (see Figure 4-8):

- Memory is connected to a bur* which consists of address lines (18 bits) and data lines (16 bits) and a read/write line (1 for read, 0 for write).

- There are many processes which wish to access memory. The processes are called bus masters; memory is called the bus slave (line 2440).

- An arbiter arbitrates between pending bus requests. There are two priority levels, 1 and 2, with 1 having priority over 2. The request and grant lines are daisy chained[1].

MemBus is essentially a simplified UNIBUS — it exhibits the synchronization of the UNIBUS while avoiding the complexities which are unneeded for our purpose. Since this is not an exercise in bus design, we have ignored timing considerations such as data skew.

---

[1] In a V-SLIDE description, we uso "DC 1o indicate that a line is to be daisy chained.

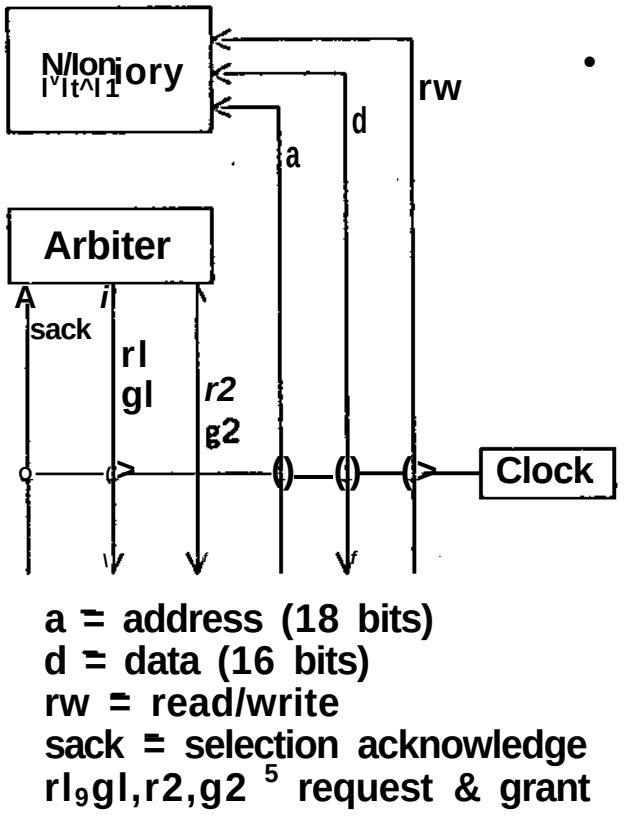


Figure 4-8: MemBus Diagram

The V-SLIDE description in Appendix I consists of three parts:

1. Module "MemBus^{H} defines the bus synchronization mechanism. It contains two synchronization capsules:

   - Synchronization capsule "Arbit" (line 280) describes the requestor protocol (roles "Requestor l^{W} and ^{H}Requestor2^{W}) and the arbiter hardware (processes "Grantor 1^{M} and "GrantorZ"). A process interfaces to the bus by assuming the role of "Requestor 1" or "Requestor2". The process must adhere to the template defined by the role.

   - Synchronization capsule "MasterSlave" (line 1410) defines the master/slave protocol for transfering data across the bus.

   Any process that assumes the roles of "Requestorl^{W} (line 860) or ^{H}Requestor2^{H} (line 1140) automatically assumes the roles of "BusUser" and "Master" also. The abstract actions "WaitBusFree", "GrabBus", and "RelBus" are defined in the ^{W}BusUser^{W} role (line 1280) which is local to the capsule.

   Hardware declared immediately within the module (lines 80 through 130 which

should be developed and this scheme should be incorporated in V-SLIDE. The problems mentioned above can not be addressed until this is done.

4.6 Conclusions

This section discusses how V-SLIDE is amenable to verification and discusses how V-SLIDE could be made a more general and practical language.

4.6.1 Meeting Our Goals

It is enlightening to see how the V-SLIDE syntax provides the facilities mentioned in the first section of this chapter. Specifically, it should provide facilities for writing verifiable descriptions and for automatic verification. These are:

1. Encapsulation: The synchronization capsule provides an encapsulation facility for synchronization mechanisms. Action definitions and hardware used by the mechanism are encapsulated within the capsule.

2. Abstraction: The synchronization capsule also provides a level of abstraction for using a synchronization mechanism. Only abstract roles and abstract action names are exported from a synchronization capsule. A process interfaces with the mechanism via these abstract roles and actions. The abstract action definitions are hidden from the process.

3. A total dosifin system: Actions used in one description (file) usually are defined in another. The correspondence between action instantiation and definition is made when a process assumes a role. At that time, it indicates which description (file) that role is defined in (see lines 1120 and 1440 of the MemBus description in Appendix I). The abstract action definition can then be accessed automatically.

4. Formally specifying the design specs: The path expression is a formal representation of the design specs. It defines which sequences of abstract action occurences are allowable.

Automatic verification will be discussed fully in the next chapters. It is important to realize that the synchronization capsule defines a synchronization mechanism completely. The processes define the synchronization hardware, and the roles define the synchronization protocol. Consequently, automatic verification can be performed on each synchronization capsule in isolation. This is important because we want the correctness of the mechanism to be independent of the various ways processes can be interconnected with the mechanism.

There is some necessary redundancy in a synchronization capsule. This comes from:

1. The desire to include the design specs in the capsule. Since these should be implied by the design (if the design is correct), they are redundant. They are,

however, needed for verification.

2. The desire to specify the synchronization mechanism completely in the capsule. This lead to the concept of roles which specify the synchronization protocol. Since all processes using the mechanism should adhere to the protocol (if they are correct), the roles are redundant. Once again, however, they are needed for verification.

3. The desire to have a clear distinction between the PE which specifies the design specs, the processes which specify the synchronization hardware, and the roles which specify the synchronization protocol.

4.6.2 Discussion of Syntax

The V-SLIDE syntax is not complete or very general. It was introduced to illustrate a number of basic concepts (encapsulation, abstraction, etc.) and could be extended in a number of ways, including:

- Adding parameters to abstract actions.
- Allowing more complex abstract action definitions with more than one statement separated by "NEXT" or ";".
- Extending the concept of a role, allowing a more general template specification.
- Adding abstract data types and combining the concepts of abstract data type capsule and synchronization capsule.

If the V-SLIDE syntax were extended appropriately, more complex examples (such as the UNIBUS) could be described easily. Also in the MemBus description in Appendix I, the address, data, and read/write lines could be encapsulated in the synchronization mechanism giving a neater and more complete description.

In our experience, specifying the PE has required a lot of thought and a good understanding of the mechanism being described. Although possibly time consuming, specifying the PE has shed some light on how our mechanism operated. This has reinforced our believe that proper structuring of the hardware description aids in the construction of correct hardware.

The V-SLIDE syntax is amenable to hardware design rather than to the documentation of existing hardware. This is because the abstractions are intended to hide the details of the design.

5. Petri Nets and Vector Addition Systems: An Introduction

This chapter introduces Petri Nets and Vector Addition Systems. Both are equivalent formalisms for control structures and can be used to represent the synchronization of concurrent processes [Petri 62] [Cooprider 76] [Hack 74] [Huen 73] [Karp 69] [Kini 75} We will define these formalisms with our uses in mind. For complete definitions, see the references. [Cooprider 76] is a good introduction to Petri Nets. [Huen 73, ChaptIV] is an introduction to Vector Addition Systems.

Later chapters will show how Petri Nets and Vector Addition Systems can be used in the verification procedure.

5.1 Standard Petri Nets

We will use the introduction to standard Petri Nets given in [Cooprider 76]:

> Petri Nets are directed graphs with two types of vertices, places (or conditions) and transitions (or events[1]). An arc in a Petri Net can connect only dissimilar vertices, that is, a place to a transition or a transition to a place. Places are usually denoted by circles, transitions by bars ...
>
> In addition, the places of a Petri Net are occupied by zero or more tokens; any allocation of tokens to the places of a Petri Net is called a marking. Often the description of a Petri Net includes the initial marking.
>
> An arc from a place to a transition designates an input place to that transition; an arc from a transition to a place designates an output place from a transition. When there is a token on every input place to a transition, it is enabled and may fire, otherwise it is disabled. If a transition fires, it takes one token from *every* input place and places one token on *every* output place.
>
> Petri Nets are interpreted by selecting sequences of firings. Any enabled transition is selected and the marking of the Petri Net altered by the rule stated above. Another enabled transition is then selected and the net marking altered again. This process is repeated indefinitely as long as there remains an enabled transition. Any marking which can be obtained in this manner is reachable from the initial marking.
>
> Note that the firing of one transition may disable another transition which was previously enabled. This can happen when two transitions share an input place; this configuration in a Petri Not is called a conflict.
>
> ... Nothing guarantees that any transition in a net must fire. However, it is useful to assume that an enabled transition will eventually fire. This assumption translates into the requirement that every process which is not blocked will be scheduled at some time and will progress.
>
> When used to represent the synchronization of concurrent processes, the

[1] We will UB» "action" not "event".

features of the Petri Net usually correspond to specific aspects of the computation. Places describe stales of processes /or conditions on data. Transitions describe actions, and the firing of a transition denotes the occurence of that action/ Tokens often denote processes, so that the "flow" of a token through the net can reflect the "progress" of a particular process. (The Petri Net does not, however, actually distinguish one token from another, so the correspondence is entirely that of the user of the net.) Other tokens represent counters or values in semaphores or messages.

5.2 Priority Petri Nets

A priority Petri Net is the type of Petri Net we will be working .with. From now on, when we refer to a Petri Net, we mean a priority Petri Net unless we explicitely state that the net is "standard".

A priority Petri Net (for our purposes) is a standard Petri Net with the following additions:

1. Weighted arcs: An arc may specify that it removes or places more than one token. This is merely an abbreviation for standard Petri Nets.

2. Self looping: A place may be both the input place and the output place of the same transition.

3. Priorities: Each transition has a priority which establishes a partial ordering between the firing of transitions. A transition may not fire if there exists some other enabled transition with a higher priority. The highest priority is 0; the lowest is oo.

4. Data places and control places: We distinguish between data places (d-places) which represent the values of data, and control places (c-places) which represent the state of a process.

5. Semantic assertions: Each transition has associated with it a (possibly null) semantic assertion which is a condition. Whenever all input c-places of the transition are enabled, the semantic assertion should be true. If it isn't, this represents an error. We will discuss this fully in Chapter 7.

6. Names: Each place and transition in the net has a (possibly null) name.

When representing a Petri Net graphically, we use bars for transitions, circles for places, arrows for arcs, and numbers for tokens (representing the number of tokens in a place). The priority of a transition ir. beside it followed by a colon (V). The name of a place or transition is written near it. If an arc is weighted, that weight is written alongside the arc. If a place is an input and output place of the same transition, a two-headed arrow is sometimes used instead of two single-headed arrows.

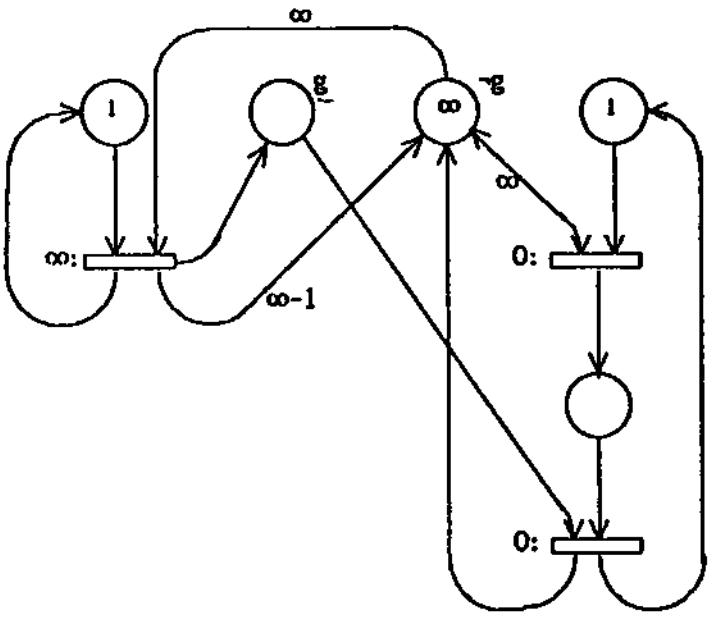


Figure 5-1: Priority Petri Net Graph

5.3 Vector Addition Systems

Petri Nets and Vector Addition Systems (VASs) are equivalent formalisms for control structures. VASs, however, are more amenable to mathematical manipulation. For our purposes, we will use a VAS as a mathematical representation of a Petri Net. The VASs we will use are capable of modeling self looping and are called "Modified Vector Addition Systems" in [Hucn 73]. First we will present the mathamatical definition of a VAS, then we will discuss the relationship between a Petri Net and a VAS in Section 5.5.

The following notation will be used when discussing VASs:

1. The relationship ≤ between two r-dimensional vectors *y* and *z* is defined as follows:

   $y \leq z$ iff $y[i] \leq z[i]$, $i=1,2,...,r$

   where integers in brackets denote components of a vector.

2. *O* or O_r represents the zero vector of r dimensions.

3. *N* represents the natural (non-negative) integers including oo[1]

[1] In this context, oo represents some arbitrarily large number such that oo-i / «o.

4. -/V represents the non-positive integers including -co.

5. The binary operator + represents addition, not logical OR.

Definition; An r-dimensional standard Vector Addition System **(VAS) V is a pair V - (m^ D) where:**

1. m_Q c N^r,

2. D is a finite set of pairs of r-dimensional integer vectors **which are called the** displacement vectors,

3. Each rf < D is a pair of r-dimensional vectors **of the form rf** « (d^1, rf^2) **where** rf^1 € $-N^r$, and d^2 (N^r.

The reachability set R(V) is the set of vectors of the form

$m_0 + \hat{}1^1 + \hat{} l^2 + \hat{}2^1 + \hat{}2^2 — + \hat{}s^1 + \hat{}s^2$

such that V i=l,2,...,s:

1. <j < D,

2. $m_0 + d\pm^1 + c*!^2 + c/2^1 \cdot \hat{}2^2 * \sim * \hat{}j^1 + rf_i^2 \ \hat{} \circ$-

5«4 Priority Vector Addition Systems

A priority Vector Addition System (PVAS) is an equivalent formalism **for** priority **Petri** Nets. This is the type of VAS we will be working with, and unless **stated otherwise, we are refering to a PVAS.**

Definition: A r-dimensional priority Vector Addition System **(PVAS) PV is a pair PV** = (m_0, D) where:

1. $\mathbf{m_0}$ **c** $/V^r$,

2. D is a finite set of triplets which are called the displacement vectors.

3. Each rf c D is a triplet of the form $d = (d^1, d^2, d^3)$ **where** d^1 (-/V^r, **and** d^2 (W, and d^3 < N (rf^3 is the priority).

The reachability set f?(PV) is the set of vectors of the form:

$m_0 + d/^1 * rf_l^2 + \hat{}2* + c/2^2 + - * \hat{}s^1 + rf_s^2$

such that V i « l,2^..f5:

1. <j < D_f

2. $m_0 + rfj^1 + </j^2\ 4\ c^{\wedge 1}\ 4\ rf_2^2 + ... + rfj^1 \bullet c/j^2 \pounds O$,

3. there does not exist *e* < D such that $c^{\wedge} < cj^3$ (higher priority) and $m_0 \bullet rf|^1 \bullet cfj^2 + c^{\wedge 1} + rf_2^2 + - + e^1 \bullet e^{2\wedge} 0$.

5.5 Creating a Priority VAS From a Priority Petri Net

This section gives an algorithm for converting a priority Petri Net into an equivalent PVAS. In subsequent chapters, we will discuss how a V-SLIDE description can be translated into a Petri Net and how verification tests can be performed on a PVAS.

A Petri Net expressed as a VAS has the following characteristics:

1. The number of places in the Petri Net is equal to the dimension of the VAS, r.

2. For each transition in the Petri Net, there is a corresponding displacement vector in D. Furthermore, the number of transitions, M, in the Petri Net is equal to the number of vectors in D.

3. For all *d* C D,

   $d^l < -/V^r$,

   and d^2 c N^r,

   and d^3 (*N.*

4. *d*^ indicates to the input places of a transition, $</^2$ indicates to the output places, and *d*^ corresponds to the priority of the transition.

5. The initial marking, MQ, of the Petri Net is identical to the vector mQ.

6. If there is a marking of the Petri Net, Mj, reachable from the initial marking, MQ, then there is a corresponding vector, mj (ft(V), and if marking M^ can be reached from Mj by the firing of transition Tj, then 3 m^ < f?(V) such that m_k = mj + tfjl + rfj^2.

Item 6. illustrates the correspondence between transitions firing in a Petri Net and vectors being added in a PVAS. This concept is central to the use of PVASs.

The following algorithm which converts a Priority Petri Net into a PVAS is a modification of Huen's PN-VAS algorithm [Huen 73, Page 148].

Algorithm PN-VAS: A priority Petri Net with r places and M transitions can be expressed as an equivalent PVAS PV - (mQ, D) of dimension r by the following steps:

1. Name the places Pj, P2,..., P_r and the transitions Tj, T21 ~9 T_M.

2. Construct the vector HIQ of dimension r as follows:

   - For *every* place Pj that is initially empty, mQ[i]=O,

   - For every place P^ that is initially nonempty, mQ[k] is assigned the number of tokens initially in place P^.

3. The set D is composed of M vectors (c/j, do?...? *d\$)* each of dimension r. For each transition Tj, a displacement vector cf^= *(dp\ d^t d^)* is constructed as follows:

   - If there is an input place Pj to transition Tj with weight w, then *d*\ [i] = -w, else rfj* = 0,

   - If there is an output place P_y from transition Tj with weight w, then rfj^fu] = w, else c/j^[u] • 0,

   - Set dk^3 to the priority of Tj.

Figure 5-2 shows the PVAS corresponding to the Petri Net in Figure 5-1.

PV « *(niQt* D)

m_Q « (1, 0, «>, 1, 0)

D - $\{d_1, d_2, d_3\}$

rfj = ((-1, 0, -00, 0, 0), (1, 1, oo-l, 0, 0), 00)
d_2 = ((0, 0, -oo,-l, 0), (0, 0, 00, 0, 1), 0)
d_3 « ((0, -1, 0, 0, -1), (0, 0, 1, 1, 0), 0))

Figure 5-2: PVAS Corresponding to Petri Net in Figure 5-1

5.6 Properties of Petri Nets and VASs

There are three basic properties of Petri Nets and VASs, reachability, liveness, and safety. Reachability has been discussed above.

We will use Huen's definitions for liveness and safety for Petri Nets [Huen 73, Page 17A]:

> A transition T of a Petri Net is <u>live</u> at a marking M if from every marking M* that can be reached from M, there exists a firing sequence which fires T. If every transition in a Petri Net is live, the net is live.
>
> A Petri Net is <u>safe</u> if none of its places ever contain more than one token at a time.

A displacement vector *d* of a VAS is live at a vector *v* if from every vector *v** that can be reached from i_f there exists an additive sequence which includes rf. If every displacement vector in a VAS is live, the VAS is live.

An r-dimensional VAS V is safe if

$$V\ v\ C\ R(V):\ v \pounds I^r$$

Current research is concerned with whether reachability, liveness, and safety of Petri Nets and VASs are decidable [Hack 74] [van Leeuwen 74] [Nash 73] [Schmid 76} For the applications we will be concerned with, reachability, liveness, and safety are decidable [Huen 73}

6. Creating Petri Nets From V-SLIDE

This chapter discusses how a Petri Net can be built from a V-SLIDE description of a synchronization mechanism. Chapter 5 discussed how a PVAS can be built from a Petri Net. Subsequent chapters will discuss how verification can be performed on a PVAS.

We will introduce basic transformations which can be applied to the parts of a V-SLIDE description to build a Petri Net. Then we will continue our previous examples of a handshake by presenting its corresponding Petri Net.

6.1 Modeling With Petri Nets

We will model a synchronization mechanism with a Petri Net. This is done by building a Petri Net from the V-SLIDE synchronization capsule which describes the synchronization mechanism. We can do this because the capsule is a complete description of the mechanism.

Since we are using a Petri Net to model a synchronization capsule, sometimes the model is not exact. We will try to point out discrepancies when they occur.

6.2 Transformations

We will introduce transformations which transform the syntactic constructs of V-SLIDE into pieces of a Petri Net. When these transformations are recursively applied to a synchronization capsule and connected, an entire Petri Net is created.

When most transformations are applied, the results "look like" a Petri Net transition to all input and output places connected to them. This is demonstrated in Figure 6-1 (a). Graphically, a yet unapplied transformation is shown as a box. Notice that when transformations are connected "box car" style (Figure 6-1 (b)), some input and output arcs must be combined on the shared transitions.

We will now discuss the transformations for the various V-SLIDE constructs used within a synchronization capsule. Believing that a "picture's worth a block of text," we will try to give more graphical examples and less rhetorical comment. Unless stated otherwise, all places are c-places, and all transitions have a priority of ∞ (lowest).

6.2.1 Hardware Declarations

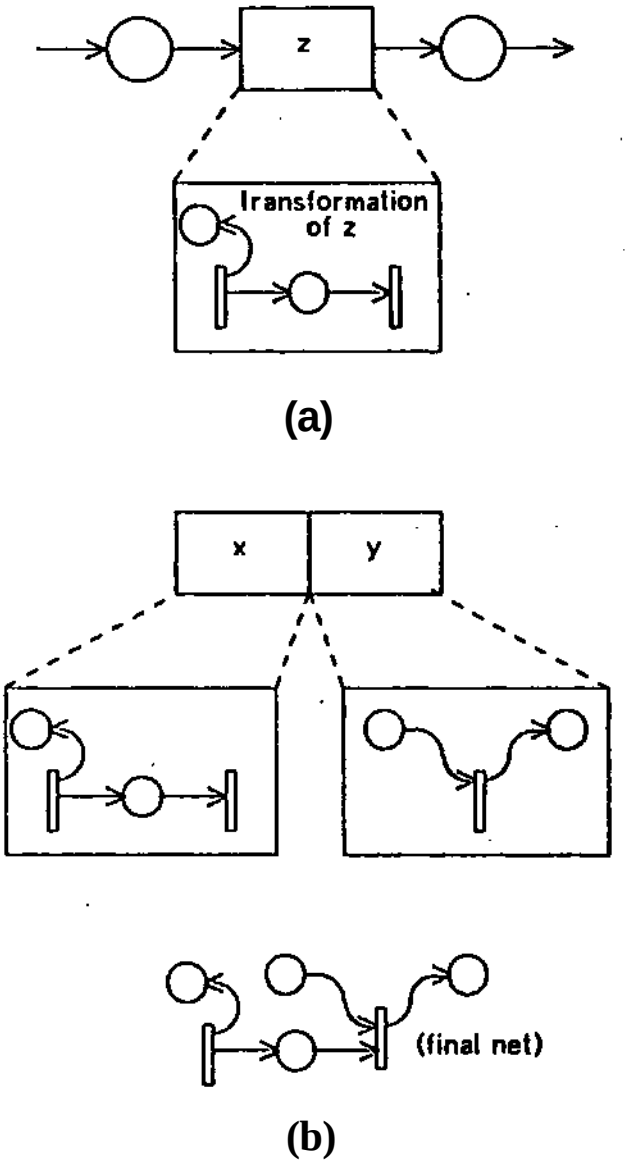


Figure 6-1: Transformation Expansion and Connection

6.2.1.1 OCAL and OCAH LINEs

TTL open collector lines have two states: 0 and 1. An OCAL (open collector active low) line is initially 0. After being set to 1 by a process, it remains 1 until that same process resets it to 0 releasing the line. We will model a single bit OC line as having two d-places: $^{w}n^{H}$ and "-n" where $^{H}n^{H}$ is the name of the line (see Figure 6-2 (a)). Initially, an OCAL line has no tokens in ^{w}n" and oo tokens in ^{m}-*vT.* An assignment of 1 to an OCAL line removes one token

from ^{H}-n^{H} and places it in ^{H}n" (Figure 6-2 (O). An assignment back to 0 does the opposite. A test to determine if the line is 1 checks for at least 1 token in "n". A test for 0 checks **for** oo tokens in ^n^{H}. OCAH (open collector active high) lines are symmetric.

Without a more detailed V-SLIDE description, DC (daisy chained) lines are difficult to model. The method chosen works correctly for the examples we have tried.

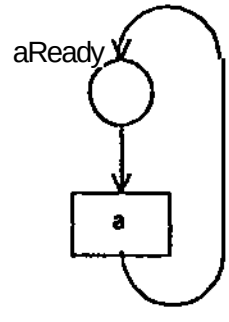
A single bit OCAL DC line is modelled as having three d-places, V, "-•n", and " n " ^{1} (Figure 6-2 (b)). Initially, there are no tokens in V or " n* " and oo tokens in %T. An assignment of 1 to the line removes one token from ^n " and places it in "n". A test to determine if the line is 1 checks for at least 1 token in "n^{H}, removes it, and places it in " n* " (Figure 6-2 (d)). An assignment of 0 to the line removes one token from "n" or " n'" and places it In *-*n" (Figure 6-2 (e)). OCAH DC lines are symmetric.

6.2.1.2 Registers and Other Declarations

Registers and other types of lines can be modelled similarly. For example, see the Petri Net model of a bit in [Cooprider 76, Page6]. Throughout the rest of this chapter, we will assume that all lines are OCAL

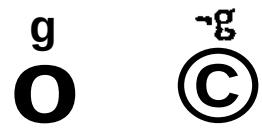
6.2.2 Processes and Roles

The transformations for processes and roles are demonstrated in Figures 6-3 and 6-4 respectively. Note that the IN1T declaration is modelled as a multi-way fork with the c-place "mutex" controlling mutual exclusion. The priorities assigned to the transitions ensure correct operation.



ROLE a; BEGIN . . . END;

Figure 6-4: Role Transformation

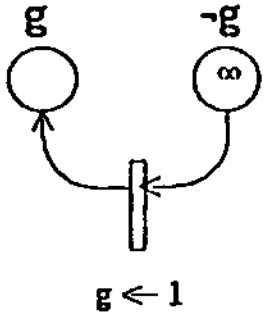


OCAL LINE g o ;

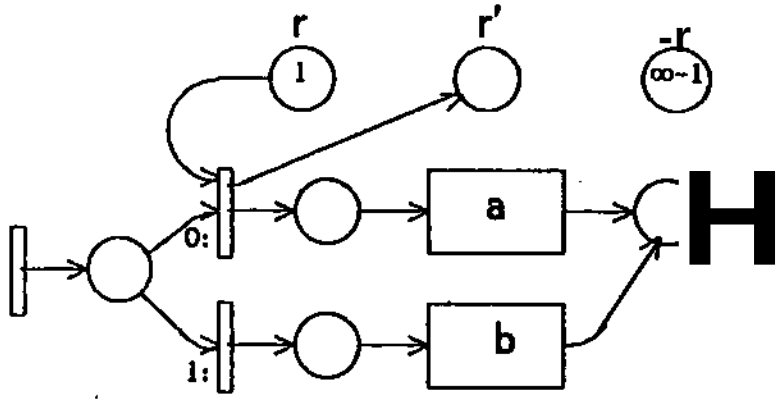
(a)

OCAL DC LINE r<>;

(b)

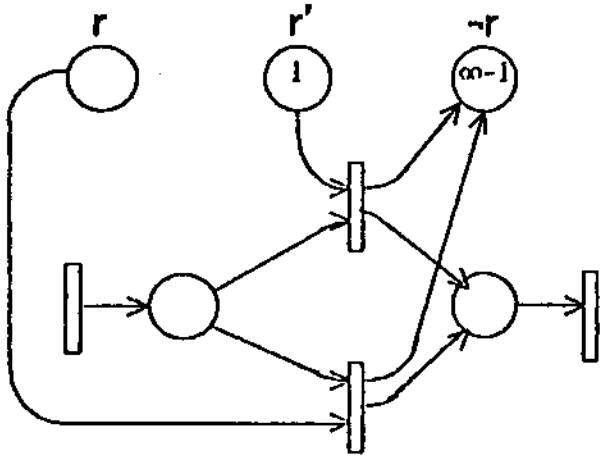


(c)



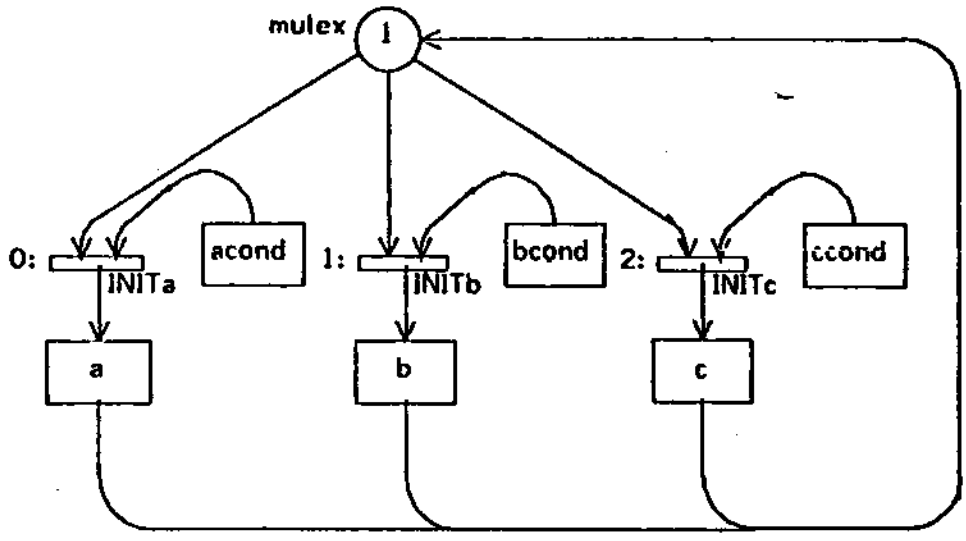
IF r EQL 1 THEN *a* ELSE b

(d)



(e)

Figure 6-2: OCAL & OCAL DC Lines



INIT a WHEN acond
ELSE INIT b WHEN bcond
ELSE INIT c WHEN ccond;

Figure 6-3: Process Transformation

6.2.3 Blocks

Any sequence of V-SLIDE statements enclosed within BEGIN and END can be decomposed into

BEGIN a NEXT b NEXT ... NEXT z END

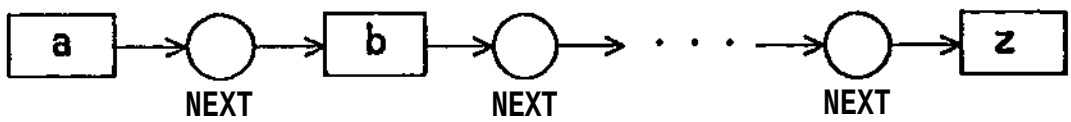
where each a...z is of the form

*5P>—5f

where each a...f is a V-SLIDE statement (which may be complex). Using this decomposition rule, the transformations for blocks of statements are given in Figures 6-5 and 6-6.

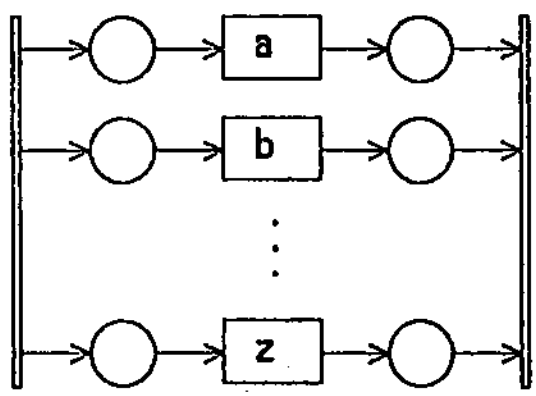
6.2.4 Abstract Actions

This section discusses the transformations for abstract Actions: the assignment abstract action and the assignment statement, the delay abstract action and the delay statement, and the conditional abstract action and the if statement.



BEGIN a NEXT b NEXT . . . NEXT z END

Figure 6-5: Block Transformation **for** NEXT



a; b; . . . ; z

Figure 6-6: Block Transformation for Semicolon

6.2.4.1 Assignment

A few words concerning the semantics of the SLIDE assignment statement are in order. If a process assigns a 1 to an OCAL line, that line remains 1 until the same process assigns a 0 to it, releasing the line. Thus OCAL lines have the "wired or" property. If a process assigns a 0 to an OCAL line, the line will not be 0 after the assignment if another process is holding the line at 1. OCAH lines are symmetric. Modelling these types of assignments was discussed in Section 6.2.1.1 and Figure 6-2.

An assignment of an upward transition ("/")* to an OCAL line has a slightly different effect

[1] Don't confuse SLIDE "upward transitions" and "downward transition*" with **Patri Net "transitions".**

than assigning a 1 to the same line. In the former case, the line is still set to 1, but it should be 0 before the assignment is made. If the line is not 0 before the assignment is made, a design error is evident. An assignment of a downward transition *C\")* has a symmetric effect. We model this type of assignment the same as before, except we give the transition a semantic assertion (see Section 5.2). Some examples are shown in Figure 6-7. Semantic assertions are shown graphically in quotes ("...") besides the transition they apply to.

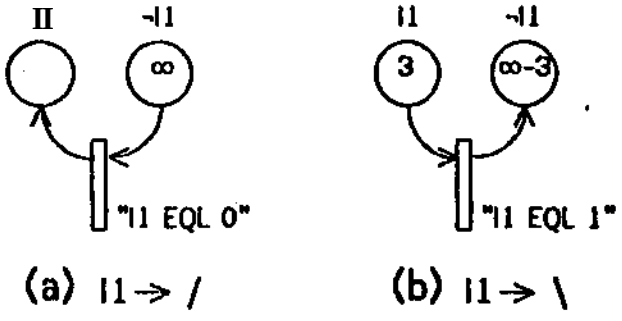


Figure 6-7: Transition Assignment Transformations

6.2.4.2 Delay

Simple delay statements of the form "DELAY .n^{H} are modelled as a single c-place (see Figure 6-8 (a)).

In the more general delay statement, the timeout clause is ignored. We do this because we are not concerned with error recovery or reliability, and we want to model worst case performance. Statements of the form

DELAY .x UNTIL cond

are transformed into

DELAY UNTIL cond

Secondly, all delay statements of the form

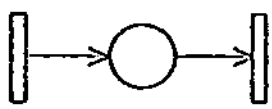
DELAY UNTIL x EQL /
or
DELAY UNTIL y EQL \

are transformed into the equivalent

DELAY UNTIL x EQL 0 NEXT DELAY UNTIL x EQL 1
or
DELAY UNTIL y EQL 1 NEXT DELAY UNTIL y EQL 0

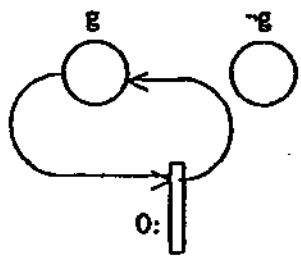
After this is done, the transformations shown in Figure 6-8 (b) and (c) can be applied. These transformations ensure that the transition does not fire until the condition is **met. All** of the transitions in the transformation are given a 0 priority (highest) to ensure **that the** transitions fire as soon as the condition becomes true.

The transformation for a more complex delay statement is shown in Figure 6-9. Here, the transformation for "cond" takes one input token and returns a token on either the "true" arc or the "false" arc.



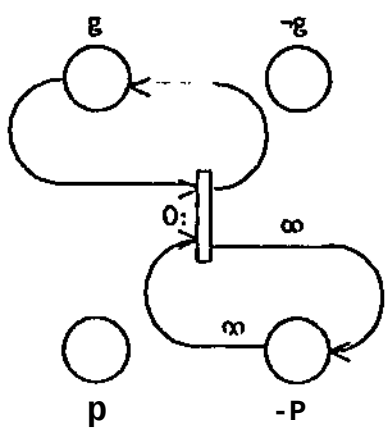
DELAY .n

(a)



DELAY UNTIL g EQL 1

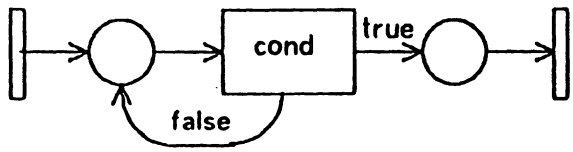
(b)



DELAY UNTIL g EQL 1 AND p EQL 0

(c)

Figure 6-8: Delay Transformations



DELAY UNTIL cond

Figure 6-9: Complex Delay Transformation

6.2.4.3 If

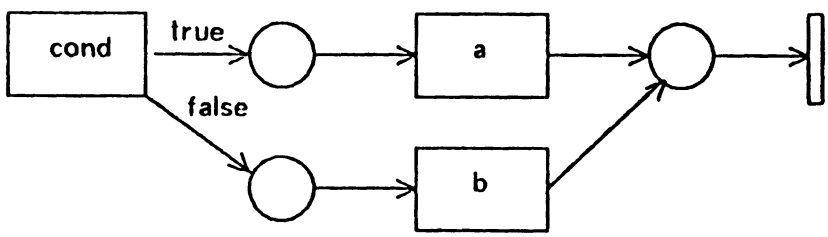
Every if statement must have an else clause before it is transformed into a Petri Net. Any statement of the form

IF cond THEN a

is first transformed to

IF cond THEN a ELSE NOP

After this is done, the transformation shown in Figure 6-10 can be applied. Here again, the transformation for "cond" takes one input token and returns a token on either the "true" arc or the "false" arc. Figure 6-11 (a) shows the transformation for the conditional "g EQL 1" where g is OCAL. Default firings can be specified easily using priorities. Sometimes, this can be used to avoid complex Petri Nets. For example, the partial nets in Figure 6-11 (a) and (b) are equivalent. However, (b) is simpler; it has less arcs.



IF cond THEN a ELSE b

Figure 6-10: If Transformation

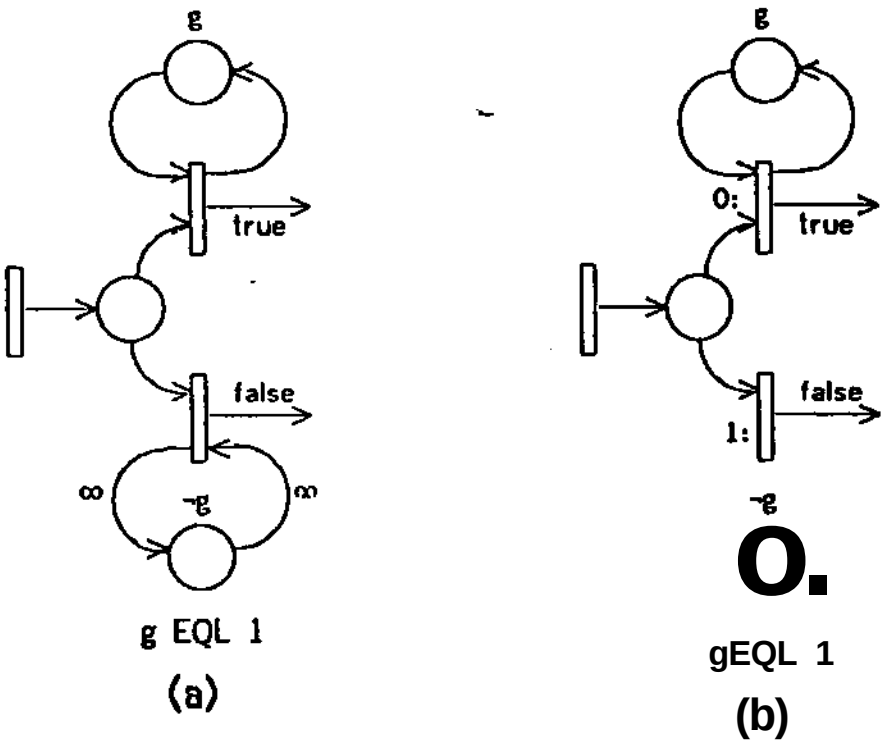


Figure 6-11: Conditional Transformation

6.2.4.4 Occurrences

When an abstract action is transformed, a transition should be given the name of the abstract action in such a way that the firing of this transition corresponds to the occurence of the abstract action. The number of times this transition has fired corresponds to the number of occurences to the abstract action. Examples for an assignment, delay, and conditional abstract actions are given in Figure 6-12.

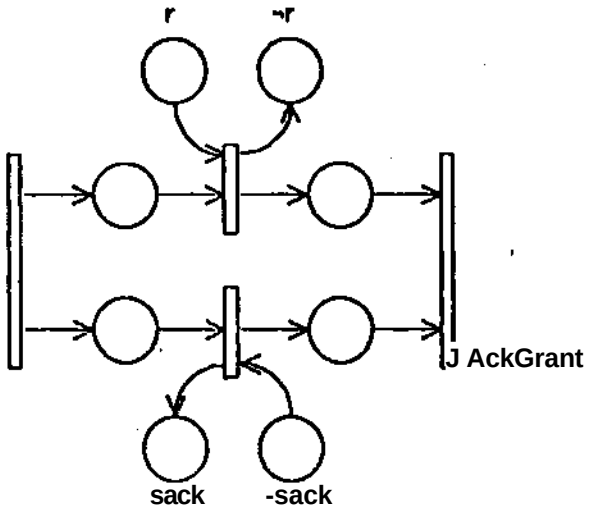
6.2.5 Other Constructs and Statemonts

6.2.5.1 Loop

The transformation for loop constructs of the form

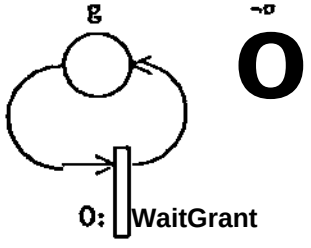
WHILE cond DO a

is shown in Figure 6-13. The "until" form is similar.



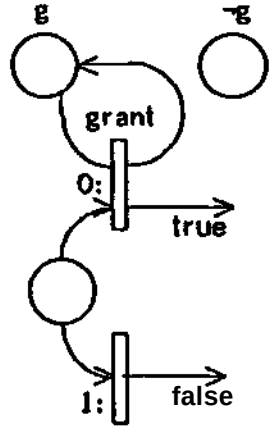
AckGrant (r <-0; sack ←1)

(a) assignment



WaitGrant (DELAY UNTIL g EQL 1)

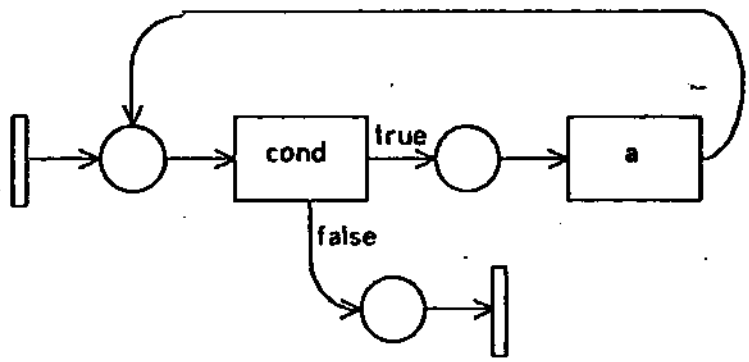
(b) delay



IF grant (g EQL 1) ....

(c) conditional

Figure 6-12: Occurences of Abstract Actions

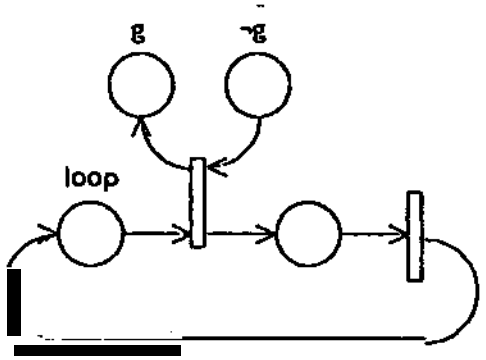


WHILE cond DO a

Figure 6-13: Loop Transformation

6.2.5.2 Labels and Branches

A branch is modelled by a transition whose only output place preceeds the statement branched to. For example, see Figure 6-14.

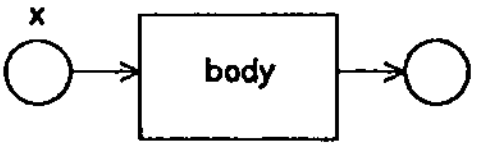


loop: g<- 1 NEXT BR loop NEXT

Figure 6-14: Branch and Label Transformation

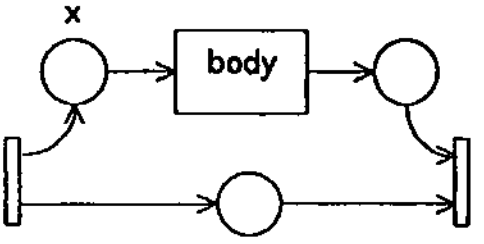
6.2.5.3 Subroutines and Calls

A subroutine is modelled as shown in Figure 6-15 (a). A subroutine call is modelled as shown in Figure 6-15 (b).



SUBR x;
BEGIN body END;

(a)



CALL x

(b)

Figure 6-15: Subroutine and Call Transformations

6.2.5.4 Nop

A nop statement is modelled as a single transition. For example, see figure 6-16.

Figure 6-16: Nop Transformation

6.3 A Simple Example - A Handshake

We will continue the example of the handshake introduced in Section 4.3 and Figures 4-5, 4-6, and 4-7. The Petri Net for the handshake example is shown in Figure 6-17. This Petri Net was built using the transformations described above. A Petri Net for the MemBus example has also been built using the transformations! but was too complex to include in this paper.

Some things worth noting about the Petri Net in Figure 6-17 are:

- The way abstract action names are given to the appropriate transitions.

- The semantic assertions used when assignments of upward or downward transitions are made to lines. If there ever comes a time in the operation of the net when all c-places to a transition are enabled, but the semantic assertion ("II EQL 0", etc.) is not true a design error is evident.

r - All transitions have a priority of co (lowest) except transitions modelling statements of the form "DELAY UNTIL cond" which have a priority of 0 (highest). This ensures that the transition fires as soon as the condition becomes true.

* - A 0 test for an OCAL line checks for no tokens in the "-»n" place; a 1 test checks for at least 1 token in the $^{H}n^{H}$ place where V is the name of the line.

- We have assumed that both roles are ready to execute and have placed one token in both places "shaker 1 Ready" and "shaker2Ready". This will be discussed more in subsequent chapters.

6.4 Optimizing the Petri Net

A few words about optimizing the Petri Net are in order. By "optimization" we mean making the net simpler: fewer places, transitions, and arcs. The verification techniques we will use in later chapters are computationally proportional to the complexity of the Petri Net obtained by the above transformations. Therefore, for complex problems, optimizing the net may be worthwhile.

We will not develop a technique for optimization, but will give one example. Figure 6-18 (a) shows the partial net obtained by using the transformations on the statements

```
r •- 0*  sack «- 1
```

Figure 6-18 (b) shows how this can be optimized into a single transition.

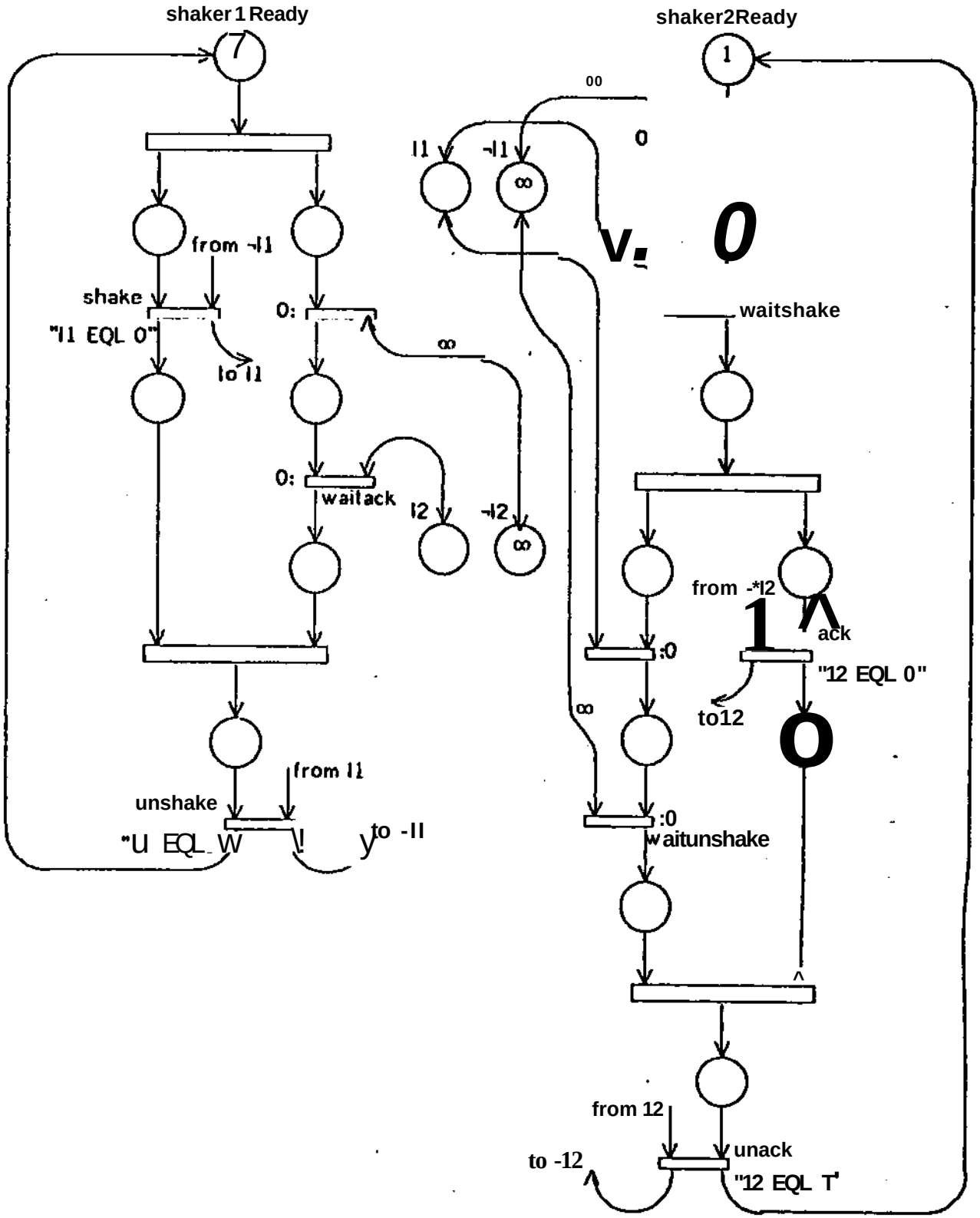
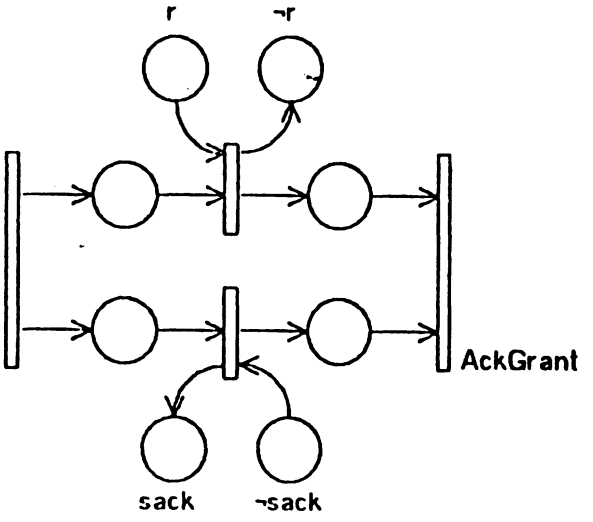
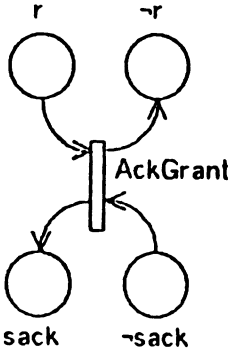


Figure 6-17: Petri Net for Handshake Example



(a) unoptimized



(b) optimized

AckGrant (r ←0; sack ←1)

Figure 6-18: Optimization Example

7. The Verification Procedure

This chapter completes the discussion of the verification procedure for V-SL1DE. We will present techniques for verifying a synchronization mechanism and for verifying processes which interface with that mechanism. We will also discuss a possible automatic verification system for V-SL1DE.

7*1 General Method

The verification procedure for a V-SLIDE synchronization mechanism consists of two parts:

1. Verifying the synchronization capsule which describes the synchronization mechanism. Specifically, we want to verify (see Section 3.2):
   - Adherence to design specs.
   - Freedom from unnecessary constraints on concurrency.
   - Freedom from deadlock.
   - Freedom from starvation.
   - Freedom from reception errors.
   - Freedom from overflows.

   A synchronization mechanism which has the above properties is considered correct.
2. Verifying that all processes which interface with the mechanism do so correctly. That is, when a processes assumes a role, we must verify that it adheres to the template defined in that role.

If we can be sure that (1) a synchronization mechanism is correct and (2) all processes interfacing with the mechanism do so correctly, then we have increased our confidence that the entire system is synchronizing correctly. We will now discuss these two parts of the verification procedure.

7.2 Verifying the Synchronization Capsule

The first part of the verification procedure is to verify the synchronization capsule which describes the synchronization mechanism. We do this by building a PVAS from the capsule and then constructing a control flow tree for the PVAS [Karp 69, Huen 73]. Various tests can be performed on the control flow tree to verify the synchronization capsule.

First we will discuss terminology and notation. Then we will present an algorithm for building the control flow tree The algorithm is from [Huen 73]. Then we will discuss the various verification tests.

7.2.1 Building tho Control Flow Tree

The following notation and terminology will be used when discussing PVAS control flow trees:

- *w* is a quantity such that V n (N:

  n < *w* and
  n + u/^{e}=u/ = u/-n.

  I.e., *w* is the classical infinity.

- A rooted tree is a directed graph such that one node (the root node X) has no arcs directed into it, each other node has exactly one node directed into it, and each node lies on a directed path from the root node X.

- If there is a directed arc from *oi* to *fl,* then *cc* is a predecessor of *fl* (or *fl* is a successor of *ot)* and we write *oc -* fl.* If the arc from *ot* to *fl* is labeled with *d,* then we write *o(. -</-* fl.*

- If od and *fl* are distinct nodes of a rooted tree, and there is a directed path from *oC* to *fl%* then we write c/ -•-» *fl.* oi is a ancestor of *fl,* and *fl* is a descendant of *ot.* Note that *oc* -» /? implies oi -»-> /?.

- If the sequence of labels of the path from *ot,* to /? is *s* « di^t-.,^, then we write:

  oi -$->-* /? or
  oi -€*!-> ... -tf$_n$-* /?.

- We shall use Greek letters (c/, *ft,* X) to denote nodes of a tree, italic lower case letters in the beginning of the alphabet (c, d_t *e)* to denote displacement vectors of the PVAS, and the end of the alphabet (p, q, 5) to denote sequences of displacement vectors.

For a description of the control flow tree we will quote from [Huen 73]:

> For a given VAS V « (mQ, D) of r dimensions, the following Algorithm T constructs a control flow tree T(V) which is a rooted tree. Each node *fl* (T(V) is labeled with an r-dimcnsional vector *lift)* where each coordinate of *lift)* is an element of N u *[w}.* The tree T(V) is a concise way of representing the states in the reachability set R(V). The initial state is represented by the label of the root node. States reachable from the initial state are shown by nodes lying on a path directed from the root node. Any state that repeats an ancestor */sic/* state is shown as a leaf of the tree T(V). Successors of such a state are not shown since they are already in the tree. There is also the case in which a sequence of

> displacement vectors can be added to a certain vector v < R(V) so that there is a net increase in some coordinates of the resulting vector v^* ... The same sequence of displacement vectors can thus be added repeatedly to v^* to generate new vectors, thus producing infinitely many reachable states. The Algorithm T indicates this situation by assigning a value w to those coordinates of v^* that have a net increase over the corresponding coordinates of the vector v.

Any state from which no further progress can be made will also be shown as a leaf of the tree T(V).

To construct a control flow tree from a V-SLIDE synchronization capsule, first create a Petri Net from the capsule (Chapter 6). Then start a number of instantiations of each role by placing tokens in their "rolenameReady" c-places (discussed in Section 7.4). If the Petri Net has M transitions and r places (re c-places and rd • r - re d-places), express the Petri Net as an r-dimensional PVA5 with M displacement vectors (Chapter 5). In doing so, name the c-places Pj, ..., P_{rc}, and name the d-places P_{rc+}j, «., P_r Consequently, all c-places will have PVAS coordinates ≤ re, and all d-places will have coordinates 2 re + 1. Also create a vector, A, which contains the (possibly null) semantic assertions for the corresponding transitions and displacement vectors. After this, construct the control flow tree using Algorithm T (below).

Algorithm T: T(V) and the labels *lift)* are constructed recursively by the following steps [Karp 69, Huen 73]:

1. Create a root node \ and label it m_0.

2. Let *ft* be the node in the tree being constructed:

   - If 3 *u* such that *</. ->-• ft* and *l{u)* « *lift),* then *ft* is a leaf of T(V), and no successors will be added to *ft.*

   - Otherwise, V *d* < D such that

     1. *lift)* + $c/^{1}$* d^2 ≥*O,* and

     2. there does not exist an e such that e^ < *d^* (higher priority) and $l(ft) + c^1 + e^2 \geq O$,

   create a successor *ft'* to *ft.* The directed arc from *ft* to /T is labeled *d.* The coordinates of /(/O are determined as follows:

   if 3 *oi* such that c/ -*-* *ft?* and *l(u)*-< *lift)* + d^l + d^2,

   then V i » l,2^..,r:

   { if *IU)[\]* < *l(ft)[\)* + d^l*l\]* + d^2*[i],*

   then Z(/O[i] is assigned the value *w*

   else /(/?')[i] is assigned the value of *l(u)[\] }*

else *l(/T)* is assigned the value of *l(fi)* + rf[1] + *d*[2].

- Also, for all *d* (D such that

  Kfi) • *d*[l] + *d*[2] • *h* ≥ *0*

  where A is a vector of re 0's followed by rd u/'s[1], if cTs corresponding semantic assertion a is not true, inform the user of a design error[2].

3. Repeat step 2. until no new nodes can be added to the tree.

[Huen 73] proves that this algorithm will terminate.

Before proceeding, we must introduce some additional terminology:

- A directed path cl. -*-»-+ *fi* in tree T(V) is called a repeatable path if (1) *fi* is a leaf of the tree T(V), and (2) *. is an ancestor of *fi* such that *l(u)* = *l(β)*.

- A number of paths pj, P2> ➡» Pm °ı a lree T ^) forms a subtree if each path pj has at least one node in common with some other path p^. The nodes and directed arcs of the subtree are the respective unions of the nodes and directed arcs of the paths.

" A maximal subtree of repeatable paths is a subtree which contains at least one repeatable path such that each repeatable path of T(V) is either entirely contained within the subtree or is completely disjoint from the subtree. Also, if any ancestor of a node is in the subtree, then that node must also be in the subtree.

We will now discuss the various verification tests.

7.2.2 Adherence to Design Specs

We want to verify that all possible sequences of abstract action occurences are allowed by the design specs specified by the PE. Each abstract action has a corresponding Petri Net transition and PVAS displacement vector. The firing of that transition or addition of that displacement vector (in step 2. of Algorithm T) corresponds to the occurence of the abstract action.

The control flow tree built by Algorithm T gives us all of the possible sequences of

---

[1] This condition checks for • transition (displacement vector) which has all c-places enabled.

[2] This corresponds to the assignment of an upward or downward transition to a line where the line is hot originally 0 (or 1).

abstract action occurences. Each directed path from the root node X to a leaf is a possible sequence. If the leaf is at the end of a repeatable path, then the sequence is infinitely long.

Verification of adherence to design specs consists of pattern matching each of the possible sequences of abstract action occurences against the PE. The techniques for this should be straight forward (e.g. see [Andler 79a] and [Andler 79b]).

7.2.3 Freedom from Unnecessary Constraints on Concurrency

We also want to verify that all allowable sequences of abstract actions are, in fact, possible. This is symmetric to verifying adherence to design specs. We expand all of the allowable sequences of abstract action occurences as specified by the PE, and match these against the possible sequences in the control flow tree. Any sequences which are not in the tree are indicative of an unnecessary constraint on concurrency.

7.2.4 Freedom from Deadlock

Freedom from deadlocks for a synchronization mechanism and liveness for a Petri Net are closely related. If the Petri Net which models the synchronization mechanism is live, then it can never be in a state such that the firing of any transition can be ruled out. Consequently, none of the actions for the corresponding synchronization mechanism can ever be blocked, and it is deadlock free [Cooprider 76, Huen 73]. The reverse is not true; a mechanism can be deadlock free, but the corresponding Petri Net may not be live. For example, there may be initialization actions which should only occur once. Nevertheless, this has not been the case for the examples we have worked with, and we can use liveness as a test for freedom from deadlocks. If the system can ever reach a state where some transitions are not live, the designer can decide if this is a design error or a property of the system.

We can test whether a Petri Net (PVAS) is live by examining its control flow tree T(V) [Huen 73} A Petri Net (PVAS) is live iff:

1. Each leaf of the tree T(V) is at the end of a repeatable path.
2. Each maximal subtree of repeatable paths of T(V) contains all displacement vectors $d < D$ as labels of its arcs.

This ensures that the Petri Net can never reach a state where the firing of any of its transitions can be ruled out. Consequently, the corresponding synchronization mechanism can never reach a state where the occurence of any of its actions can be ruled out.

We distinguish between a displacement vector (transition or action) being deadlocked and

being critically deadlocked. If there exists a leaf of the control tree which is not at the end of a repeatable path, then all displacement vectors are deadlocked at that node (state). For each maximal subtree of repeatable paths of T(V), any *d i* D which is not a label of one of its arcs is deadlocked from all the nodes in the subtree. For each node of the tree ©d, let *dl(u)* be the set of displacement vectors (transitions or actions) which are deadlocked from that state. Then for each *d* < $dl(u)_f$ if 3 i *= l,...,rc such that *l(u)[i]* + cf*[i] ≥ 0, then *d* is critically deadlocked. That is, a critically deadlocked displacement vector (transition or action) at node *ot* is deadlocked and has at least one of its input c-places enabled. This corresponds to an action in the system which is partially (but not fully) enabled.

The distinction is made because a critically deadlocked transition is very indicative of a design error. Any initialization actions or the like will be shown to be deadlocked but not critically deadlocked.

7.2.5 Freedom from Starvation

We can test whether a starvation problem exists by examining the control flow tree T(V). If there is a repeatable path which does not contain all displacement vectors *d* C D_f then it is possible that the system could remain on this path, and a starvation problem exists. All displacement vectors (transitions or actions) not on the repeatable path are starved (we can ignore those which are deadlocked). Once again, it may be worthwhile to distinguish between starvation and critical starvation.

7.2.6 Freedom from Reception Errors

If a Petri Net is safe, then no part of the net receives a token until the previous token has been handled. This is an indication that the corresponding synchronization mechanism is free of reception errors. Other reception errors (such as lost signals) will show up as a deviation from the design specs or as a deadlock. We can test if a Petri Net is safe by examining its control flow tree [Hucn 73} A Petri Net is safe iff the label of every node *u* of the control flow tree T(V) has components ≤ 1.

For our purposes, we are only concerned with the c-places, not, the d-places which could have any number of tokens in them. If any node *u* of the control flow tree T(V) has a component J(c/.)[i] > 1 (where i •= l_r..,rc), then this may indicate a reception error.

7.2.7 Froodom from Ovorflow Errors

The limited formalism we have developed can not be used to detect overflow errors. We have ignored the problems of modelling FIFO buffers and iterative loop constructs with Petri Nets. If the modelling techniques were extended appropriately, then it could be possible to test for overflow errors by examining the control flow tree. We will leave this for future work.

7.3 Verifying the Role Instantiations

Once the synchronization capsule has been verified and is correct, we must ensure that all processes which interface with the mechanism described by the capsule do so correctly. That is, each process which assumes a role (called a role instantiation) must adhere to the template defined by that role.

This can be verified by pattern matching the body of the process against the template defined by the role. The pattern matching is performed by applying the following transformations to the process body. These steps are done for each process and for each role assumed by that process (one at a time).

1. Discard any statements and any actions not defined within the current role.
2. Collapse redundant semicolons and "NEXT"s.
3. Transform any occurence of ^{H}a NEXT BEGIN b END NEXT c" into ^{M}a NEXT b NEXT c^{H}.

The result should match the current role definition exactly.

For an example, we will continue our discussion of MemBus. If we want to ensure that process ^{H}ClockProcess^{W} adheres to the template defined in the role "Requestor1", we apply the above transformations to the body of "ClockProcess". The result is shown in figure 7-1. Here, the match with the role definition for ^{W}RequestorI^{H} is exact except for the "NEXT" after the action "AckGrant" in "ClockProcess". In this case, the error is a minor one which the designer may choose to ignore.

At first, it may seem that a "NEXT" could be used by a process where a V was used in the role's template. In this last example, in fact, this was true. However, it is not true in general as demonstrated by Figure 7-2 where the process has used a "NEXT" between action "Req* and action "WaitGrant" instead of the semicolon. Here, this is a serious design error since the use of "NEXT" may cause the grant signal to be missed. This is because the definition of

```
BEGIN
   Req: WaitGrant NEXT
   AckGrant NEXT
   WnitBusFree NEXT
   GrabBus NEXT
   WaitDropGrant NEXT
   UnackGrant NEXT
   ReI Bus
END
```

Figure 7-1: Verifying "ClockProcess"

"WaitGrant" is "DELAY UNTIL gI EQL /'[1].

```
BEGI N
   Rcq NEXT WaitGrant NEXT
   AckGrant NEXT
   WaitBusFrco NEXT
   GrabBus NEXT
   WaitDropGrant NEXT
   UnackGrant NEXT
   ReI Bus
END
```

Figure 7-2: Incorrect Use of "NEXT"

7.4 V-SLIDE Verification System

In this section we will discuss a possible automatic verification system for V-SLIDE. This should tie together many of the concepts introduced so far.

As mentioned in Chapter 3, we see verification as a necessary and natural step in the design cycle. Consequently, the verification system should be easy to use.

The user runs the verifier and specifies the name of the module (file) which is to be verified. If the module contains a synchronization capsule, a Petri Net is built. For each role defined within the capsule, the user is asked the number of instantiations which are to be started. This number of tokens is put in the "rolenameReady" c-place in the Petri Net (e.g.

[1] Of coureo Iho definition is not visible out Bide Ihe "Arbil" capsule, so making • decision on using "NEXT" versus V based on the definition would subvert the verification procedure. Consequently, the role should be adhered to exactly unless the actions are totally unrelatod.

the c-place named "aReady" in Figure 6-4). Then the PVAS and control flow tree are constructed, and the various verification tests are performed (Section 7.2).

If a design error is uncovered, the verifier should provide a trace of the abstract action occurences which lead up to the error. This trace can be read directly from the tree T(V).

If any processes within the module assume a role, correct interfacing should be verified by the technique discussed in Section 7.3. Any discrepancies should be pointed out to the user.

For an example, we will use our MemBus description in Appendix I. We want to verify the "MemBus" module and the modules for all processes which interface with the bus (e.g. modules "Clock" and "Memory"). A hypothetical dialog is shown in Figure 7-3. (User input is underlined.)

7.5 Discussion

In this section, we will discuss the problems and advantages of the verification methods used.

7.5.1 Problems

One major problem with these techniques is modelling V-SLIDE with Petri Nets. The modelling of open collector and daisy chained lines discussed in Chapter 6 may not always be correct. Modelling more complex structures such as arrays, buffers, etc. is ad hoc. To offset this disadvantage slightly, we are concerned with a restricted environment, the synchronization capsule. The use of complex structures in a synchronization capsule is uncommon. However, if the concept of a synchronization capsule was extended, this may no longer be true.

Another problem is the complexity of Petri Nets. Nets of even modest complexity become unwieldy and graphically unintelligible. This is a moot point though since all handling of Petri Nets is done by machine; complexity is irrelevant. However, graphical feedback may be handy.

A final problem is the size of the control flow tree. Hack [Hack 74] has suggested that the size of the tree is on the order of Ackerman's function applied to the number of transitions and places of the corresponding Petri Net. This indicates that optimization of the Petri Net (Section 6.4) may be an important step. From our experience, an example similar to MemBus had a control flow tree of moderate size (about 100 nodes). Handling a tree an order of magnitude larger than this should not be a problem if an "efficient" language such as BLISS [Wulf 71] is used.

```
•L verify
V-SLIDE VERIFICATION SYSTEM

Module to verify: McmDus
[Using file "MEMBUS.SL1")
[Verifying SYNCHRONIZATION CAPSULE "Arbit"
How many instantiations of role "Rcquestorl"! 2.
How many instantiations of role "Requestor2"i $}
Results:
        •
        •
        •
]

[Verifying SYNCHRONIZATION CAPSULE "MasterSlave*
How many instantiations of role "master"J 1
How many instantiations of role "slave"! 1
Results:
        •
        •
        •
]

Module to verify: Clock
[Using file "CLOCK.SLI"]
[Verifying roles
Process "ClockProcess" does not adhere to template defined in
        role "Requestorl" in module "MemBus".
        Discrepancy is ...
Process "ClockProcess" adheres to template defined in role
        "Busllser" in module "MemBus".
Process "ClockProcess" adheres to template defined in role
        "Master" in module "MemBus".
]

Module to verify: Memory
[Using file "MEMORY.SLI "]
[Verification of roles
Process "Memory" adheres to template defined in role
        "Slave" in module "MemBus".
```

Figure 7-3: Verification Dialog for MemBus

The designer of the verification system should make a tradeoff decision. He can verify the

[1] The verifier won't ask about role "BusUcer" because this role is local to the capsule.

control flow tree one path at a time and conserve memory. Or he can build the entire tree and save execution lime by combining paths of the tree which repeat another path. From our experience, much of the control flow tree will be duplicated Consequently, the later decision may be best since the memory cost may be small while the execution time savings may be large.

7.5.2 Advantages

The one overriding advantage of the techniques discussed is that they are quite amenable to automation. Not only is the designer encouraged to structure his design with V-SLIDE, but also automatic verification is as painless as running a program.

8. Contributions of This Paper

This paper has made four contributions to the area of communications link design by hardware descriptive languages. First, we defined a synchronization mechanism and identified its two closely related parts: the synchronization hardware and the synchronization protocol.

Second, we identified the types of verification we would like to perform on a synchronization mechanism. These are:

- Error recovery.
- Reliability.
- Adherence to design specs.
- Freedom from unnecessary constraints on concurrency.
- Freedom from deadlock.
- Freedom from starvation.
- Freedom from reception errors.
- Freedom from overflow.

Third, we introduced a syntax which was amenable to writing verifiable hardware descriptions and to automatic verification. This syntax included the concepts of encapsulation, abstraction, and abstract action definition and instantiation. We also introduced the concepts of a synchronization capsule which describes a synchronization mechanism, and a role which describes a synchronization protocol.

Forth and last, we developed a formalism which could be used to automatically verify synchronization mechanisms. This formalism involved translating hardware descriptions into Petri Nets and then to Vector Addition Systems. Verification tests could then be performed on the control flow tree for the Vector Addition System.

The ideas expressed in this paper could be implemented together or separately. In particular, the implementation of a language similar to V-SL1DE would aid the designer by allowing him to structure his design properly. Also, the verification techniques described could be applied to any hardware descriptive language which allows synchronizing processes (e.g. SLIDE), although V-SL1DE would be more amenable to this.

Additional Comments

After review of the work presented in this paper, it has been brought to our attention that the path expressions (PEs) used in this paper can be simplified considerably while maintaining all verifiable properties. We took the attitude that all abstract action names must appear in the PE. If this restriction is relaxed, we find that most delay abstract actions (we have usually named these "wait...") can be removed from the PE. The PE still specifies the design specs, but does so in a cleaner and more understandable way. The order of occurences of abstract actions should be enforced by the roles and processes since they still use the delay, abstract actions.

To demonstrate this, the module for the bidirectional handshake discussed in Section 4.3 and shown in Figure 4-5, Page 16, is shown in Figure 8-1. The only changes have been made in the PE where the delay abstract actions have been removed. The resulting PE is much more understandable than the original.

The PEs in the MemBus description in Appendix I could be simplified similarly.

```
MODULE handshake;

   SYNCHRONIZATION CAPSULE shake (ROLES shaker1, shaker2)j

        PATH
           (shaker1.shake  NEXT
            shaker2.ack  NEXT
            shnkcrl.unshakc  NEXT
            shnker2.unack)*
        END;

        OCAL LINE ll<>, I2<>;

        ROLE shakcrl:
          BEGIN
            shake (II <- / ) ;  waitack (DELAY UNTIL 12 EQL /) NEXT
            unshake (II «- \)
          END:

        ROLE shaker2:
          BEGIN
            waitshake (DELAY UNTIL II EQL /) NEXT
            ack ( 1 2 - / ) ; waitunshake (DELAY UNTIL II EQL \) NEXT
            unack (12 «- \)
          END:

   END lof synchronization capsule shake];
END lof module handshake];
```

Figure 8-1: Handshake Module (Modified)

I. V-SLIDE Description of MemBus

LI Module MemBus

```
 10 MODULE MemBus;
 20
 30      !
 40       The following declarations are considered global and are
 50       available to any process which assumes any of the roles
 60       defined in this module.
 70      !
 80      OCAL LINE
 90              a<17:0>,          ! memory address lines (18 bits)  !
100              d<15:0>,          ! memory data lines (16 bits)  !
110               rwo;             ! read/write line !
120
130     MACRO     ! values for the read/write line !
131               read  := 18,
132               write := 88;
140
150     !
160       The following capsule describes the bus arbitration
170       mechanism.  Bus masters interface with the bus by assuming
180       roles of "Requestorl" or "Requestor2".  Doing this, they
190       automatically assume the roles of "BusUser" and "Master"
200       also.  Memory interfaces with the bus by assuming the
210       role of "Slave".
220
230       The role names "Requestorl" and "Requestor2" as well as
240       all abstract action names are exported from the module.
250       The role name "BusUser" as well as the lines "rl", "r2",
260       "gl", "g2\ "sack", and "bbsy" are local to the module.
270     !
280 SYNCHRONIZATION CAPSULE Arbit (ROLES Requestorl, Requestor2);
290
300     PATH
310        LET   ! pending requests !
320              prl = //(Requestorl.Req)  - #(Grantorl.Grant),
330              pr2 = //(Requestor2.Req)  - #(Grantor2.Grant)
340         IN
350
360               Requcs tor1.Req*;
370               Requestor2.Reqv<;
380
390               ((Grantorl.Grant [prl>B] NEXT
400                 Requestorl.UaitGrant NEXT
410                 Roquestori.AckGrant NEXT
420                  (Grantorl.UaitAckGrant NEXT Grantorl.DropGrant);
430                  (Requestorl.BusUser.UaitBusFree NEXT
440                          Requestorl.BusUser.GrabBus) NEXT
450                 Requestorl.UaitDropGrant NEXT
460                 Requestorl.UnackGrant NEXT
```

```
470               Grantorl.UaitUnackGrant)
480
490             +(Grantor2.Grant [pr2>0 AND prl-B] NEXT
500               Requestor2.UaitGrant NEXT
510               Requestor2.AckGrant NEXT
520                (Grantor2.UaitAckGrant NEXT Grantor2.DropGrant);
530                (Requestor2.BusUser.UaitBusFree NEXT ^
540                        Requestor2.BusUser.GrabBus) NEXT
550               Requestor2.UaitDropGrant NEXT
5G0               Requestor2.UnackGrant NEXT
570                Grantor2.UaitUnackGrant))*;
580
590              (BusUser.UaitBusFree NEXT
G00               BusUser.GrabBus NEXT
Gl0               BusUser.Re I Bus)*
820     END ! of path !?
630
640      !
650       The following declarations are encapsulated.
660      !
670     OCAL LINE        ! The request lines !
680              rlo_f r2<>;
690
700     OCAL LINE        ! The daisy chained grant lines !
710             gl<>, g2<>;
720
730     OCAL LINE        ! The selection acknowledge line !
740              sacko;
750
760     OCAL LINE        ! The bus busy, line !
770              bbsyo;
```

```
790     INIT Grantor1 WHEN r1
800     ELSE INIT Grantor2 WHEN r2;
810
820     !
830      The following role defines the actions which a process
840      at priority level 1 must follow to access the bus.
850     !
860     ROLE Requestor1 (ROLES BusUser, Master);
870        BEGIN
880             Req (r1 ← 1);
890                     WaitGrant (DELAY UNTIL g1 EQL /) NEXT
900             AckGrant (sack ← /; r1 ← \);
910                     WaitBusFree NEXT
920             GrabBus NEXT
930             WaitDropGrant (DELAY UNTIL g1 EQL 0) NEXT
940             UnackGrant (sack ← \) NEXT
950             RelBus
960        END;
970
980     !
990      The following process defines the granting mechanism
1000     for priority level 1.
1010    !
1020    PROCESS Grantor1;
1030       BEGIN
1040            Grant (g1 ← /);
1050                    WaitAckGrant (DELAY UNTIL sack EQL /) NEXT
1060            DropGrant (g1 ← \);
1070                    WaitUnackGrant (DELAY UNTIL sack EQL \)
1080       END;
1090
1100    !
1110     The following role defines the actions which a process
1120     at priority level 2 must follow to access the bus.
1130    !
1140    ROLE Requestor2 (ROLES BusUser, Master);
1150       BEGIN
1160            ...similar to Requestor1...
1170       END;
1180
1190    !
1200     The following process defines the granting mechanism
1210     for priority level 2.
1220    !
1230    PROCESS Grantor2;
1240       BEGIN
1250            ...similar to Grantor1...
1260       END;
1270
1272    !
```

```
1274      The fol lowing role defines the protocol for
1276      grabbing the bus.
1278     !
128B     ROLE DusUser;
1298        BEGIN
1308             UaitBusFree (DELAY UNTIL NOT bbsy) NEXT
1318             GrabBus (bbsy «- /) NEXT
1328              RelBus (bbsy •- \)
1338        END;
1348
1358 END ! of synchronization capsule arbit !;
```

```
1378     !
1388      The following capsule defines the master/slave protocol
1398      for transfer ing data across the bus.
1408     !
1418 SYNCHRONIZATION CAPSULE fiasterSlave (ROLES Master, Slave);
1428
1438     PATH
1448              (Master.UaitSIave NEXT
1458               Master.ShakeSIave NEXT
1468               Slave.UaitShakeSlave NEXT
1478               Slave.AckMaster NEXT
1438               Master.UaitAckMaster NEXT
1498               Master.UnshakeSlave NEXT
1588               SIave.WaitUnshakeSIave NEXT
1518               Slave.UnackMaster)>v
1528     END ! of path !;
1538
1540     OCAL LINE       ! master and slave synch lines !
1558             msyno, ssyno;
15G8
1578     ROLE Master;
1588        BEGIN
1598             WaitSlave (DELAY UNTIL ssyn EQL 8) NEXT
1688             ShakeSIave (msyn «- / ) ;
1610                     WaitAckMaster (DELAY UNTIL ssyn EQL /) NEXT
1628             UnshakeSlave (msyn «- \)
1638        END;
1640
1658     ROLE Slave;
1668        BEGIN
1678             WaitShakeSlave (DELAY UNTIL msyn EQL /) NEXT
1688             AckMaster (ssyn «- / ) ;
1698                     WaitUnshakeSIave (DELAY UNTIL msyn EQL \) NEXT
1780             UnackMaster (ssyn «- \)
1718        END;
1728
1738 END ! of synchronization capsule MasterSlave !;
1748
1758 END I of module MemBus !;
```

1.2 Module Clock

```
1760 MODULE Clock;
1770
1780 PROCESS ClockProcess (ROLES MemBus.Requestorl);
1790
1800     !
1810      Local registers,
1820     !
1830     REGISTER ClockReg<15:0>, ClockAddr<17:8>;
1840
1850     BEGIN
18G0              .
1870              .
1880
1890        !
1900             Now we make a bus request and wait for a grant.
1910             Then we ack the grant.
1920        !
1930        Req;  WaitGrant  NEXT
1940        AckGrant  NEXT
1950
19G0        !
1970             Now we wait till the bus is free and then grab it.
1980        !
1990        UaitBusFree  NEXT
2000        GrabBus  NEXT
2010
2020        !
2030             Now we start two parallel sections of code.
2040             The first waits until the grant is dropped, and then drops
2050             the acknowledgement.  The second does a read-modify-wri te
20G0             operation with memory.
2070        !
2080        BEGIN
2090             UaitDropGrant  NEXT
2100             UnackGrant
2110        END;
2120
2130        BEGIN
2140             a «- ClockAddr;  rw «- read  NEXT ! Set up for read cycle !
2150             UaitSlove  NEXT                   ! Wait till slave is ready
21G0             ShakeSlove;                       ! Handshake with slave ...
2170                     Uai tAckfiaster  NEXT     ! and wait till slave acks
2180
2190             ClockReg «- d;                      ! Read data ... !
2200                     UnshokeSlave  NEXT        ! and finish handshake !
2210
2220             ...modifyClockReghere.
```

```
2238           d «- ClockReg;  ru •- write  NEXT ! Set up for write cycle !
2240             UaitSlave  NEXT                 ! Wait till slave is ready
2250             ShakeSlave;                     ! Handshake with slave ...
2268                    Wai tAckMaster  NEXT     ! and wait for ack !
2278             UnshakeSlave  NEXT              ! Finish handshake !
2288           d «- 0;  a «- 0;  rw •- B              ! Release Mne9 !
2298        END  NEXT
2388
2310        !
2328             Now release the bus.
2338        !
2348       Re I Bus
2358             •
23B8             •
2378     END;
2388
2390 END ! of module Clock I;
```

1.3 Module Memory

```
2400 MODULE Memory;
2410
2420    INIT MemProcess UHEN TRUE;
2430
2440    PROCESS MemProcess (ROLES MemBus.Slave);
2450
2460            !
2470             Memory (16 bit).  The top address is set by
2480             the simulation time parameter ".TopMem".
2490            !
2500            CMOS REGISTER m[B:.TopMem]<15:B>;
2510
2520       BEGIN
2530            WHILE TRUE DO
2540               BEGIN
2550                     !
2560                      Uait until the handshake cycle is started by the
2570                      master, and then do the read or write operation.
2580                     ;
2590                     UaitShakeSlave  NEXT
2600                     IF ru EQL read
2610                        THEN d «- mta]
2620                        ELSE mla) «- d  NEXT
2630
2640                     !
2650                      Now ack the master and uait for the master to
2660                      drop the handshake.
2670                     !
2680                     AckMaster;  UaitUnshakeSlave  NEXT
2690
2700                     !
2710                      Now release the data lines and finish the
2720                      handshake cycle.
2730                     !
2740                     d «- 0;  UnackMaster
2750               END
2760            END;
2770
2780 END ! of module Memory !;
```

References

[Andler 79a] **Sien Andler.**
Predicate Path Expressions.
In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages* , pages 226-236. ACM, January, **1979.**

[Andler 79b] **Sten Andler.**
Predicate Path Expressions: A High-Level Synchronization Mechanism.
PhD thesis, Carnegie-Mellon University, to be submitted, 1979.

[Ansi 79] **Ansi Technical Committee X3T9; Minicomputer Project 54.**
U.S.A. Contribution to ISO TC97/SC13 for a Small Computer to Peripheral Bus Interface Standard.
December 1979, revision 2.

[Barbacci 78] **Mario Barbacci, et. al.**
The ISPS Computer Description Language.
Technical Report, Carnegie-Mellon University **Computer Science** Department, March 1978.

[Barbacci 75] **Mario Barbacci.**
A Comparison of Register Transfer Level Languages for Describing Computers and Other Digital Systems.
IEEE Transactions on Computers C-24(2):, February 1975.

[Burr 79] **William E. Burr, et. al.**
A Bus System for the Military Computer Family.
Computer 12(4):11, April 1979.

[CHDL 75] **ACM, IEEE, New York.**
International Symposium on Computer Hardware Descriptive Languages, 1975.

[Chen 74] **Robert Chia-Hua Chen.**
Bus Communications Systems.
PhD thesis, Carnegie-Mellon University, January, 1974.

[Cooprider 76] **Lee W. Cooprider.**
Pctri Nets and the Representation of Standard Synchronizations.
Technical Report, Carnegie-Mellon University Computer Science Department, January 1976.

[Habermann 69] **A. Nico Habermann.**
Prevention of System Deadlocks.
Communications of the ACM 12:373-377, July 1969.

[Hack 74] **Michel Hack.**
The Recursive Equivalence of the Reachability Problem and the Liveness Problem for Petri Nets and Vector Addition Systems.
In *15th Annual Symposium on Switching and Automata Theory* , pages **156-164. IEEE, October, 1974.**

[Huen 73] Wing-Hing Huen.
A Unifying Notation and Analysis of Modular Register Transfer (RT) of Control.
PhD thesis, Carnegie-Mellon University, December, 1973.

[Ichbiah 79] J. D. Ichbiah, et. at.
Rational for the Design of the ADA Programming Language.
ACM SIGPLAN Notices 14(6):part B, June 1979.

[IEEE 78] IEEE.
Special Issue on Fault-tolerant Digital Systems.
Proceedings of the IEEE :, October 1978.

[Karp 69] R. M. Karp and R. E. Miller.
Parallel Program Schemata:.
Journal of Computation and Systems Sciences 3(4):147-165, May 1969.

[Kini 75] Vittal Kini.
A System to Test for Safcncss and Liveness of RT-Level Hardware Designs.
Technical Report, Carnegie-Mellon University Computer Science Department, August 1975.

[Levy 78] John Levy.
Buses, The Skeleton of Computer Structures,
In Gorden Bell, et. al., *Computer Engineering - A DEC View of Hardware Systems Design.* Chapter 11. Digital Press, 1978.

[Nash 73] B. 0. Nash.
Reachability Problems in Vector Addition Systems.
American Mathematical Monthly 80:292-295, 1973.

[Parker 78] Alice C. Parker and John J. Wallace.
The Developement of GLIDE: A Hardware Descriptive Language for Interfacing and I/O Port Specification.
Unpublished: available from the authors.

[Parker 79a] Alice C. Parker, et. al.
The CMU Design Automation System: An Example of Automated Data Path Design.
In *Proceedings of the 16th Design Automation Conference* , pages 73-80. ACM, IEEE, June, 1979.

[Parker 79b] Alice C. Parker.
High-Level Language Constructs for Handling Input/Output.
Unpublished: available from the author.

[Petri 62] C. A. Petri.
Fundamentals of a Theory of Asynchronous Information Flow.
In *Proceedings of IFIP* , pages 166-168. IFIP, 1962.

[Schmid 76] H. A. Schmid and E. Best.
Towards a Constructive Solution of the Liveness Problem in Petri Nets.
Technical Report, Universitat Stuttgart - Institut Fur Informatik, April 1976.

[van Leeuwen 74] Jan van Leeuwen.

A Partial Solution to the Reachability-Problem **for Vector-Addition** Systems.

In *6th Annual Symposium **on** Theory **of Computing*** 9 **pages 303-309. ACM, May, 1974.**

[Wallace 79] John J. Wallace and Alice C. Parker.
SLIDE: An I/O Hardware Descriptive Language.
In *International Symposium **on Computer Hardware Descriptive Languages*** .ACM, IEEE, October, 1979.
Accepted for publication.

[West 78] C. K West.
General Technique for Communications Protocol Validation.
IBM Journal of Research and Development 22(4):, **July 1978.**

[Wulf 71] W. A. Wulf, et. al.
BLISS Reference ManuaL
Technical Report, Carnegie-Mellon University Computer Science Department, October 1971.
Revision 4.