UNDERSTANDING DIGITAL SYSTEM SPECIFICATIONS WRITTEN IN NATURAL LANGUAGE

by

John Joseph Granacki, Jr.

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(Electrical Engineering)

December 1986

UNDERSTANDING DIGITAL SYSTEM SPECIFICATIONS WRITTEN IN NATURAL LANGUAGE

by

John Joseph Granacki, Jr.

Technical Report CRI-87-02

A Dissertation Presented to the

FACULTY OF THE GRADUATE SCHOOL

UNIVERSITY OF SOUTHERN CALIFORNIA

In Partial Fulfillment of the

Requirements for the Degree

DOCTOR OF PHILOSOPHY

(Electrical Engineering)

December 1986

UNIVERSITY OF SOUTHERN CALIFORNIA
THE GRADUATE SCHOOL
UNIVERSITY PARK
LOS ANGELES, CALIFORNIA 90089

This dissertation, written by

................JOHN JOSEPH GRANACKI, JR.......

under the direction of his....... *Dissertation Committee, and approved by all its members, has been presented to and accepted by The Graduate School, in partial fulfillment of requirements for the degree of*

DOCTOR OF PHILOSOPHY

.........Barbara Solomon.........
Dean of Graduate Studies

Date ....December 9, 1986....

DISSERTATION COMMITTEE

.........Alice C. Parker.........
Chairperson

....

....

DEDICATION

In memory of my father

ACKNOWLEDGEMENTS

I would like to thank my advisor, friend and colleague, Alice Parker who kept me focused, provided key insights, guidance and encouragement. Her enthusiasm and good humor buoyed me up through the rough passages.

I would also like to thank Yigal Arens, especially for PHRAN which made the prototype system a reality. Yigal also provided valuable critiques and comments which helped sharpen the ideas presented in the thesis.

I thank Mel Breuer who introduced me to the subject of design automation. He provided special insight and valuable suggestions which enhanced the presentation of this research.

I thank John Choma, Dennis Mc Leod and John Silvester who served on my doctoral committee.

I thank my wife, Evelyn, for the untold sacrifices she made so that I could pursue this research. She provided the nurturing and care without which I would have never finished. No words can express my appreciation.

I thank Jeannette Wing who provided much of the stimulus for the development of the semantics of the model of behavior presented in this work.

Thank you, David Knapp, my friend and office-mate for lunches at CA Pizza and Pasta Co. and the great technical discussions.

Thank you, Nohbyung Park for your interest in my work and the

good times we shared and Fadi Kurdahi for your listening, encouragement and help with Scribe.

At USC, I have made many friends and enjoyed my time with the DA group. Thanks go to Sally Hayati, Mitch Mlinar, Jorge Pizarro and Esther Brotoatmodjo for their friendship and help.

Over the years, I have always received tremendous encouragement from my good friends, Eric Cave and Angelo Checki--thanks guys.

I have received support and encouragement from the management of Hughes Aircraft Company and my friends and colleagues, there, including my managers, Jack Frantz, Bob Little, and Frank Schmid and also Bill Spehar and Jay Block.

In the past several years, Bob Brown has been a mentor as well as a great lunchtime companion--thank you Bob.

My special thanks to Tim Ramey: Tim always seemed to ask the right questions and spent a significant number of hours listening to my results and forcing me to probe deeper.

I am also indebted to Hughes for the use of their fine technical library and document center and their cheerful and helpful staff, particularly, Debbie, Regina, Marsha and Dorothy.

Thanks to our two cats Brie and Moët who sat on top of my computer and kept me company in the wee hours of the morning.

Finally, I am grateful to all those not specifically mentioned who helped me in this endeavor.

ABSTRACT

This thesis concerns itself with the specification of digital systems. The specific focus of the work described here has been on understanding system specifications written in natural language. The long term goals of the research are to provide methods and software to assure that the specifications are consistent, correct, and complete.

The research described here differs from previous research in several ways. First, the natural language input is used to construct an internal design representation, rather than just to query about existing design data. Second, using natural language allows a generality of expression not found in formal models. Finally, the natural language is not overly restricted.

A major part of the research described here involves formally modeling the information found in system specifications. An extension of the USC Design Data Structure is described, with emphasis on timing and control flow. Then, this extension is used to model various concepts found in system specifications, such as unidirectional value transfers and temporal constraints. These models then provide a basis for the templates against which input specifications are matched.

A semantic parser, PHRAN, is used as the basis for the actual interface software. PHRAN contains a knowledge base of sentence patterns along with associated concepts. PHRAN inputs English sentences and looks for patterns in the sentences. When it finds a pattern match, the concept associated with the pattern is particularized with the information found in the sentence.

After PHRAN has parsed the input, the SPAN (SPecification ANalysis) package constructs fragments of the design data structure described above, and informs the user what design information has been found.

PHRAN-SPAN currently contains 13 concepts, 100+ nouns, and 25 verbs. It can handle ambiguity, nouns used as modifiers, and verbs used as nouns. It has processed a number of sentences which come from actual specifications.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

1.1. The Problem

The general digital system specification problem is that of capturing and formally representing the *necessary and sufficient* information to implement hardware, firmware and software which perform a prescribed function in a well-behaved and predictable manner while satisfying any constraints and achieving acceptable performance. The input to the specification process is a set of requirements that describe the *real world* functions to be carried out by the system. The digital system specifier translates these into the domain of digital systems and adds required details as necessary to insure that the system's *behavior* is correct. In practice, specification of digital systems is generally done in an informal manner, relying heavily on written prose and various types of diagrams, *e.g.*, timing diagrams and flowcharts. Informal specifications are subject to a number of recurring flaws, including but not limited to missing information, incorrect information, overspecification and inconsistencies. Since these four subproblems seem to be the most prominent, we address them individually here.

1.1.1. Missing information

The primary problem with informal system specifications is that there is no method for determining whether the information provided is complete. By *complete* we mean that it is possible to produce a correct implementation from the specification information. However, no general model of abstract behavior suitable for system specification exists in the published literature; therefore, there is no framework on which to base definitions of the necessary or required information for completely specifying a system's behavior. The functional model of behavior has been used in some cases, but this simply requires that the proper inputs and outputs and the desired transformations be defined. Though this is trivial, it has proven useful in various semi-formal specification systems [Heninger 80], [Tiechroew 77] and will be incorporated in the techniques proposed here.

1.1.2. Incorrect information

Correctness of the information is a more difficult problem than completeness. In general, correctness requires either an alternate or high-level specification to compare against, additional information (*e.g.*, assertions), or some other form of redundant information like typing of variables. Alternate specifications may contain the same flaw(s) as the original or primary specification, or different styles associated with the specifications may make comparison difficult. Furthermore, there is no complete formal behavioral representation which can be used in the verification process. Limited aspects of the specification may be verified independently by using techniques such as Petri Net simulation or temporal logic theorems to prove certain properties of the system, but these techniques are not easily extensible to the full range of behavior required.

1.1.3. Overspecification

Overspecification implies that the designer has restricted the range of implementation unnecessarily. As in the case of missing information, the ability to detect overspecification is critically dependent on the existence of a general model of behavior. This model is required to generate the criteria for nonessential information. Frequently, the overspecification results because the designer specifies an implementation, *i.e.*, the *how to do it* rather than the *what to do.* This problem is exacerbated by lack of the proper constructs in the language for specifying the desired behavior and/or by language constructs which force the specifier to use an implementation style or level of description that is inappropriate.

1.1.4. Inconsistencies

There are two aspects of consistency that must be considered [Winchester 80], one of which can be considered syntactic and one semantic. The first is exemplified by checking for the numbers of inputs and outputs, and ensuring that such things as item names, and description formats are the same throughout the specification. Semantic conflicts of information, such as expecting different behaviors from different states which actually are identical, are more difficult to detect.

1.1.5. Other deficiencies

Though some formal specification languages attempt to address some aspects of the four problems cited, the resulting specifications often lack clarity and are incomprehensible to all but the language experts.

Finally, existing specification languages do not allow the expression of causality except as a side effect of an actual implementation. The ability to express causal behavior without forcing an implementation is essential when specifying control information.

1.1.6. Lack of an adequate behavioral model

To understand the discussion in this section and Section 1.4, a definition of a *process* is provided.

Definition 1.1: A process is an independently executing activity.

In system specifications, a process can: be started asynchronously (whenever specified conditions become true); execute indefinitely; start, suspend and terminate other processes asynchronously; exclude other processes from executing; communicate with other processes; and be asynchronously terminated or suspended itself when some specified conditions become true. The (clock) rates at which these processes run may be different from process to process (*i.e.*, not a multiple of any common fundamental clock). Processes communicate via shared data, synchronize at critical points, or compete for shared resources. This behavior is described in more detail in Section 1.4. The problem with specifying system behavior is that the behavior of multiple communicating and competing asynchronous processes is not well characterized by existing models. This is even true when the communication is across parallel branches in a single process and the events are not completely ordered. Most existing techniques for describing or specifying these types of processes overly restrict the solution, resulting in a structured, more costly hardware design; they simply cannot be used to specify the behavior of complex systems composed of large numbers of components and many levels of hierarchies.

Having defined the general digital system specification problem, we will now identify the specific aspects of the problem to be addressed by this research. This work will demonstrate that an informal description of the behavior of a digital system written in English using a restricted vocabulary can be used to produce a formal representation of that system's behavior.

We will show that the formal representation produced in this way will allow us to check for certain missing information. Prior research [Parker 83] has shown that this formal representation of the system's behavior can be checked for the correctness of temporal storage and the transfer of values. This work will also demonstrate that certain forms of ambiguity that arise in natural language descriptions can be detected using a few simple heuristics and corrected through interaction with the user.

The remainder of this chapter is organized as follows. Section 1.2 presents the proposed goals of this research. Section 1.3 discusses the overall approach to this research. Section 1.4 identifies four classes of asynchronous behavior and describes each class in detail as well as giving examples of the kinds of sentences that are used in describing this behavior. Section 1.5 discusses the representation and mapping issues associated with the problem and how they influenced our approach. Finally, Section 1.6 provides an outline to the remainder of this thesis.

1.2. Proposed Goals of this Research

In this section, we will first present our long term goal which forms the framework for this research. Then we will describe a set of subgoals corresponding to the results presented in this thesis.

The long term goal of this research is to produce a system with the capability to accept and analyze an informal specification for a digital system.

The system will be capable of analyzing the specification to detect the four properties:

- incompleteness,
- ambiguity,
- inconsistency, and
- redundancy.

Ideally, the system would be interactive and would assist the user in the construction of the specification. Before a system of this type can be built, several open research problems must be solved. They include research into

1. man-machine dialogues,
2. formal models of digital system behavior,
3. knowledge representation, and
4. system specification.

Because of the broad scope of these problems, subgoals were formulated for the immediate research. The subgoals are

1. to produce a system that accepts a large number and variety of single sentences that are characteristic of informal specifications,
2. to provide a system that is capable of analyzing the input for completeness and ambiguity, and
3. to construct a system that maps the natural language input into a formal model of digital system behavior.

The approach outlined in Section 1.3 is intended to produce a system satisfying these subgoals. The future problems not included in the subgoals or approach will be discussed in Chapter 7.

1.3. The Approach

The goal of this research is to model, using a small set of system-level concepts and the design data structure, the information to be found in a system specification. This research is based on the premise that informal specifications are useful (albeit error-ridden) and that natural language will continue to be the primary representation tool of the system designer. Therefore, our approach is to provide the system designer with an automated tool that accepts *restricted* English text as input and uses this input to construct a *formal* model of the behavior specified. This formal model may then serve as an input to other tools which perform design synthesis, and analysis, including validation and verification. If the formal model is *neutral*[1] and *complete*, the information contained in the model can be transformed into other formal models to allow the full power of techniques like data flow analysis, Petri nets and temporal logic to be used when appropriate. Even if a formal system specification model were desired the research performed here would be a prerequisite to such a model.

To be effective, this interface tool must not corrupt the specifier's input in any way. However, it must resolve potential ambiguities as well as assisting the user to produce a correct, complete, consistent and comprehensible specification.

The following steps were followed in developing this specification technique:

1. characterize the system specification problem,

2. classify the information contained in specifications into a small set of concepts,

---

[1] By neutral, we mean that there is no implementation-bias (*e.g.*, a preferred synchronization scheme like monitors or a scheme that requires the user to explicitly denote potentially parallel processes).

3. determine how to apply past or related research to the current problem,
4. develop a model of abstract system behavior,
5. use existing parsing software to construct a prototype natural language interface, and
6. validate the specification technique via a set of well-chosen examples.

1.4. Classes of Behavior

Behavior of a digital system may be synchronous or asynchronous. The model of behavior and the specification technique investigated in this research support the specification of both kinds of behavior. However, since synchronous behavior is better understood, this research has focused on asynchronous behavior, particularly asynchronous concurrent behavior.

1.4.1. Four Classes of Asynchronous Behavior

Asynchronous concurrent behavior is prevalent in I/O interfaces and other internal interfaces between two separate, independently clocked systems. This asynchronous concurrent behavior may be further classified into one of four categories:

1. asynchronous branching,
2. interprocess control and subprocesses,
3. communicating sequential processes, and
4. competing processes initiated asynchronously.

1.4.1.1. Asynchronous branching

The asynchronous branching class includes all dynamic escape mechanisms, *i.e.*, the process may enter a different state at any instant of time. Hardware *resets* are a common example, as well as the *timeout* mechanism used in displays, terminals, pocket calculators, bus and network controllers and certain fault-tolerant systems.

1.4.1.2. Interprocess control and subprocesses

The interprocess control and subprocess class illustrates the various mechanisms required by one process to control another subordinate process, *i.e.* subprocess. These mechanisms are: initialize, start, wakeup or enable, suspend or inhibit, and terminate. These mechanisms are thought to be a complete set. An example of this is a network controller process which initiates a subprocess which performs a selective receipt of messages sent in a broadcast mode.

1.4.1.3. Communicating sequential processes

The class of communicating sequential processes involves at least two processes which are proceeding concurrently and are required to share access to data or exchange messages as an integral part of their activity. This situation is common to operating system theory, which contains a rich set of examples. A UART (Universal Asynchronous Receiver/Transmitter) device which contains one process reading data at one rate and a second process accessing this data and writing it out at a different rate is an example of this class of behavior.

1.4.1.4. Competing processes initiated asynchronously

The class of competing processes initiated asynchronously is the result of asynchronous branching by two or more concurrent processes to two processes or states that require mutual exclusion in time. This class occurs when handling concurrent exceptions that are never *expected* to occur concurrently. Conflicts over the flow of control or over resource sharing can occur when unexpected concurrent execution occurs. For example, an error condition in an arithmetic pipeline stage which occurs concurrently with a cache page fault may interact in an unpredictable or undesirable fashion. Several independent studies have shown this to be a major source of errors in production systems [Parker 83], [Suzuki 84].

1.4.2. Kinds of Sentences

Sentences describing systems with these four classes of behavior were taken from actual specifications [USN 73], [IBM 74], [DEC 79], [AdvancedMicroDevices 80], [Intel 84]. The ability to accept these sentences demonstrates the potential practical utility of this specification technique. Examples of these sentences are provided in the following list:

1. *A block of data bytes is transferred by a sequence of data cycles.*
2. *The peripheral equipment shall sample the EF code word which is on the OD lines.*
3. *Each requestor communicates with the arbiter via two lines, a request line and a grant line.*
4. *When a requestor needs the shared resource, it asserts its own request signal to the arbiter.*
5. *Upon receipt of the assertion of INTR, the arbitrator ceases to issue BGs.*
6. *Select shall be dropped 100 ns after the write is begun.*

7. *Reset terminates any operation in progress, and clears the status register to zero.*

8. *When read of an External Register begins, the EB Read/Write line shall be raised.*

9. *The data handshake cycle is controlled by the TX and TYA lines with the line CX=0.*

10. *During the request phase, the requesting agent places address and control information onto the bus.*

11. *Agents will assert the BUSERR* line whenever they detect a problem with data, address, or control information.*

Having characterized the classes of behavior and the kinds of sentences we expect the system to process, we now examine the representation issues raised in Section 1.3.

1.5. Representation and Mapping Issues

To process a specification written in natural language requires at least a target representation and a procedure to map the text into the target representation. We partitioned the problem of mapping the restricted English input into the final representation of the design data into two parts:

1. Mapping from English sentences into a semantic model of the domain of discourse, and

2. Mapping from the semantic model of the domain of discourse to the internal model of digital system behavior.

This partition has two advantages:

1. It reduces the problem of mapping across a large *semantic gap* into two simpler mapping problems.

2. It permits the mapping from English to the intermediate representation to be done on one sentence at a time.

The disadvantage is that two representations and two mapping procedures are required to solve the problem. The interrelationship of the two representations is discussed in Chapter 4. Each representation will be introduced individually here.

1.5.1. Model of the Domain of Discourse and Mapping

Various representation schemes used by natural language understanding [Schank 73], [Schank 75], [Schank 81], [Tennant 81], [Dyer 83], [Hayes 83], [Sowa 86] and other areas of artificial intelligence [Minsky 68], [Bobrow 75], [Findler 79], [Schank 77], [Sowa 84] were reviewed in developing the underlying representation used to model the domain of discourse. The style of representation developed was modeled on Schank's Conceptual Dependency (CD) formalism. This style was chosen because it is

- declarative,
- easily extended to a new domain,
- based on common action verbs,
- based on the notion of causality, and
- used by a large number of parsers.

For the mapping process from English to the semantic model, PHRAN (PHRasal ANalysis) [Arens 86] was chosen because:

- it is a knowledge-based natural language parser,
- its knowledge representation is based on CDs, and
- it is documented and available.

1.5.2. Model of Digital System Behavior and Mapping

The choice of a model for digital system behavior was based on the following criteria:

1. a formal definition of the semantics,
2. neutrality, *i.e.* no implementation bias,
3. the ability to capture causal relationships,
4. a complete timing model for both synchrony and asynchrony,
5. the ability to support incomplete designs,
6. the ability to support hierarchy, and
7. the ability to represent other types of design information (in addition to behavior).

As with the model for the domain of discourse many models were reviewed. Several candidates satisfied five or six of the seven criteria; however, no representation satisfied all seven. Many of the models suffered from implementation bias or only supported synchronous behavior or asynchronous behavior. Most of the alternatives are discussed in Chapter 2 as related research. Additional information and pointers to the literature can be found in several texts on concurrent and distributed computation, programs and modelling [Cohen 86], [Gehani 86], [Filman 84], [Paker 83], [Kahn 79].

The mapping procedure from the semantic model of the domain of discourse to the model of digital system behavior is performed by a separate program called SPAN for SPecification ANalysis. The problem addressed by SPAN is greatly simplified by the tight coupling between the semantic model of the domain of discourse and the model of behavior. SPAN identifies major structures in the model of behavior and their missing components. SPAN also identifies ambiguity and can alert the specifier as to the source of the ambiguity. SPAN's final task will be to combine the pieces of the specification created on a sentence by sentence basis into a single representation. This synthesis task is discussed in Chapter 7 under future research.

1.5.3. The Semantics of the Design Data Structure

The DDS was designed for representing digital designs in a synthesis system like the USC Advanced Design Automation System (ADAM) [Granacki 85]. However, the semantics of the DDS—in particular, the semantics of the representation of timing and sequencing information—were not fully defined. Therefore, as part of this research we defined the semantics of both the data flow subspace and the timing and sequencing subspace as discussed in Chapter 3. We also extended the timing and sequencing model to cover causal relationships, constraints and delays by refining the basic representation. The refinements and extensions to the model led to a better understanding of asynchronous behavior. With these improvements the model could satisfy the criteria for representing digital system behavior enumerated in Section 1.5.2.

1.5.4. Extensions to PHRAN

PHRAN's basic capability for "parsing" natural language input are quite extensive and most of the basic words, phrases and sentences only required additions to PHRAN's database. However, some problems arose in parsing noun phrases and also nouns and verbs with the same lexical stem (*e.g.*, address, clock, interrupt, process, transfer).

In both these cases, modifications were required to PHRAN's control routine and also generalized syntactic patterns were required to support the parsing. The details of these extensions are discussed in Chapter 5.

1.6. Thesis Outline

Chapter 2 describes related research. This research touches on a number of different areas: computer hardware description languages, programming languages, mathematical models for hardware and software, graphical techniques, specification of programs and software systems, and informal techniques. Based on this review, the deficiencies in existing representations and techniques are identified.

Chapter 3 describes the DDS, which was selected as a *formal* model of abstract system behavior. This description focuses on the semantics of the Data Flow Subspace and the Timing and Sequencing Subspace and the *bindings* between these two subspaces. The other two subspaces, the Structural Subspace and Physical Subspace, are described only when they are required to complete a behavioral description.

Chapter 4 presents the relationships between the natural language input and the various representations used by PHRAN-SPAN in processing this input. PHRAN's output is the input to SPAN, the analysis program. Example sentences are given for each type of abstract system behavior and its corresponding DDS template. The natural language basis for the DDS templates is also described.

Chapter 5 describes the components of the natural language interface: the corpus (a collection of writings), PHRAN and the additions to PHRAN and its knowledge base to process system specifications. Also, an input format based on the IEEE standard for Software Specifications is presented.

Chapter 6 presents results of running test cases. The results consist of both successful and unsuccessful attempts at understanding sentences. The unsuccessful attempts are analyzed and discussed in detail along with possible future approaches to the problem.

Chapter 7 summarizes the conclusions reached in performing this research and future research problems to be considered.

Chapter 2

Related Research

2.1. Introduction

There is a large body of research which has addressed the problem of describing digital hardware systems, commonly known as CHDLs (Computer Hardware Description Languages). In addition, various areas in computer science research have focused on the problems of specifying software systems and programs and describing concurrent behavior. These techniques may be divided into two distinct categories, formal and informal specifications.

2.2. Formal Languages

Most formal languages for hardware description involve register-transfer behavior. Only a few have intended to capture concurrent asynchronous behavior.

2.2.1. SLIDE

SLIDE (Structured Language for Interface Description and Evaluation) [Parker 81] is a language designed for the description of input, output, interfaces, and interconnections. It is based on the concept of a process and allows the description of asynchronous concurrent processes which can communicate with, compete with, and initiate and terminate other processes. SLIDE provides mechanisms for delay and timeout and other I/O specific functions, *e.g.*, formatting and FIFO buffers.

SLIDE also allows the designer to specify technology-relative behavior of the hardware lines and registers.

SLIDE addresses a level of abstraction which is slightly broader in scope than a register-transfer level language. In fact, the SLIDE syntax is a superset of the ISPS (Instruction Set Processor Specification) syntax; therefore SLIDE is partially constrained by ISPS' capabilities and also suffers from some of the same problems, *e.g.*, mixing behavioral and structural information. In addition, the process interactions must be described at least partially in terms of their hardware implementation.

Despite these shortcomings, SLIDE is one of the few CHDLs which permits description of communicating asynchronous concurrent processes. Parker and Wallace identify some important primitives for modeling this type of hardware. The research proposed here is aimed at levels of abstraction above the one available in SLIDE and also may propose extensions to SLIDE such as those in SLIDE+ [Parker 84].

2.2.2. Other HDLs

In a tutorial on CHDLs, Shiva [Shiva 79] listed forty-three languages. Since then more than 20 new languages have appeared in Ph.D. dissertations [Matty 83], [Huang 81], [Kumar 82], [Moore 82], [Singh 81], [Dudani 80] and other publications [Uehara 83], [Koomen 85], [Barbacci 85]. Some of these languages are low level and address only logic-level descriptions, while others represent only incremental enhancements or successors to already powerful languages. There are a few exceptions like VHDL (VHISC Hardware Description Language) which doesn't belong to either of these two categories and will be discussed in Section 2.2.2.4.

Most of these languages have been developed to support simulators [Lipovski 78], and some are just variants of programming languages. Those which will be discussed here were selected as being representative of a particular class of languages and demonstrate important concepts in describing or specifying hardware.

2.2.2.1. ISPS

Instruction Set Processor Specification (ISPS) [Barbacci 79a] is one of the most widely used CHDLs. ISPS is a procedural language for describing the behavior of computer hardware at the register-transfer level. It has been used extensively for description and evaluation of different architectures in the Military Computer Family effort [Barbacci 77] [Barbacci 79b]. ISPS is also the input language for describing the behavior of complex digital systems to the CMUDA system [Parker 79a] which synthesizes hardware. Other uses have involved the generation [Nagle 81] and verification [van-Mierop 78] of microcode, and educational uses [Parker 79b].

The lack of constructs for expressing timing and the lack of process level constructs, particularly those required for cooperating processes as described in Section 1.4, make ISPS a good but incomplete model of behavior at the register-transfer level. Therefore, ISPS cannot be used directly as a specification language or model of behavior for this research.

2.2.2.2. DDL

DDL (Digital system Design Language) [Duley 68], [Uehara 81] is an example of a block-oriented nonprocedural language. More importantly, DDL requires some units of hardware to exist and be named [Dietmeyer 74]. There is some ability to describe timing but it is at a very detailed level and does not support clock variables. The general underlying description of sequencing within a block is described by a finite automaton. Some hierarchy can be accommodated by linking the state machines from one level to another through a control variable.

Though its nonprocedural nature is very powerful and allows description of asynchronous branching, it requires the specification of too much implementation-oriented detail to serve as a specification language.

2.2.2.3. ADLIB

ADLIB (A Design Language for Indicating Behavior) [Hill 79] is an example of a multilevel language. It is intended to model the architectural level, the register-transfer level, gate level and circuit level. ADLIB is strongly linked to SDL (Structural Description Language) [vanCleemput 77] and is only used to represent the behavior of the components whose interconnection is specified by SDL. As a superset of Pascal, ADLIB allows the very powerful recursion constructs of Pascal and essentially mixes the description of software and hardware in these components. ADLIB is a very powerful language for describing a system to a simulator but requires the sequencing be described in detail through timing relations.

ADLIB introduces some useful control structures like UPON and TRANSMIT, but its use of generalized programming constructs make it difficult to determine what hardware is specified and what software is specified. This is likely to result in inefficient solutions to both parts of the problem. The reflection of the structure of the modules in the behavior will tend to produce overspecified designs.

2.2.2.4. VHDL

VHDL (VHSIC Hardware Description Language) [Dewey 84], [Shahdad 85], [Saunders 85], [Intermetrics 85], [Nash 86] represents a significant advance in state-of-the-art hardware description languages. VHDL was strongly influenced by two requirements:

1. to use Ada constructs whenever possible, and
2. to only include features in the language that can be realized in hardware.

These requirements resulted in VHDL being the first hardware description language to employ the package concept originating in Ada which provides a mechanism for encapsulating definitions and utility functions. This feature can be used to encapsulate technology dependencies into one location. Another result was the incorporation of strong typing and user-defined data types, a feature that gives the engineer the capability of defining convenient abstractions (such as "instruction" or "address") and the operations associated with abstractions [Dewey 84].

VHDL does not support a simulation process model, *i.e.*, it cannot suspend (wait) sequential statement execution and then continue based on passage of time or occurrence of some condition. VHDL does not support wire delays or global time and it prohibits the use of global variables, making it difficult to describe and model clocking schemes. Other controversial aspects are discussed in the VHDL critique of Nash and Saunders [Nash 86]. Furthermore, they state that "Some of these issues have been discussed by VHDL developers; however, no semantically consistent solution was found satisfactory during VHDL's design."

With a formal language as complex as VHDL, some language design decisions may arbitrarily exclude the ability to describe behavior in a neutral way, thus resulting in an implementation bias.

2.2.3. Specification Languages for Software

There have been several attempts to develop specification techniques for software development [Gehani 86]. Two of the most notable are Clear [Burstall 81] and Gist [Goldman 82]. Clear is based on algebraic theory; namely, the work of Goguen *et al.* [Goguen 77], [Goguen 79]. Gist is a formal language with an ALGOL-like syntax which permits expressibility by the provision of many of the constructs found in natural language specification of processes. Each of these languages will now be discussed in the context of this research.

2.2.3.1. Clear

In Burstall's own words "...the primitive operations of Clear are very close to the underlying mathematical theory and they are not as powerful as one might desire for convenience of expression. Perhaps Clear could be thought of as an assembly language, though one with procedures and user definable types. We hope at some future time to provide higher level languages based on the same semantic ideas, which will be of greater practical value in software engineering."

This summary demonstrates the two main deficiencies of the algebraic specification method --- the difficulty of construction and the lack of comprehensibility. Another deficiency in adapting or building on these techniques for hardware specification is their current lack of concurrent description techniques and their inability to specify performance [Liskov 79].

2.2.3.2. Gist

Gist, on the other hand, aims at solving many of the same problems addressed by this research; however, it differs in two important ways. The first is scope; Gist makes no assumptions about what is being specified and hence, requires a closed specification that includes the environment. London and Feather [London 82] point out that "Dealing with the distinction between system (the portion of the specification to be implemented) and environment (the remaining portions of the specification which establish the framework which the system will operate) is very difficult." This is very important in that the end use of our specification is synthesis and we are not going to synthesize the environment. The second difference is the ability to represent both synchronous and asynchronous behavior. In trying for generality, Gist is built on the asynchronous notion of demons, making it relatively poor at synchronization [Cohen 84]. Also its concept of histories would not support the level of detail necessary to specify timing for the hardware in a digital system.

One of the features of Gist reflected in this work is the declarative representation of constraints. No examples of Gist available in the literature are applicable to the domain of this research.

2.2.4. Programming Languages

Using programming languages for describing hardware limits the choice to those languages which can describe concurrent behavior. Some possible candidates are Concurrent Pascal, MODULA 2 and Ada. All are modern languages with modularity, powerful data abstraction mechanisms and structured programming environments. The basic difference between them is their mechanism for process interaction [Andrews 83]. C-Pascal and MODULA 2 are procedure-oriented, monitor-based languages, whereas

Ada is an operation-oriented language which uses remote procedure calls. Ada's form of interaction is related to procedure-oriented process interaction as well as message-oriented languages like Communicating Sequential Processes[2] [Hoare 78]. Therefore Ada enjoys the advantages of both types. Each of the languages mentioned restricts the user to describe the problem in a different way, but all can be shown to be equivalent descriptions.

As in the case of ADLIB, these languages will probably result in overspecified designs. As Andler [Andler 79] indicates, this level of description still involves implementation details and does not support specification of a design. Though these three languages are not directly applicable to the hardware specification problem, they offer valuable insight into the general specification problem. A good example of this insight comes from another programming language for distributed systems, NIL [Strom 83]. This language uses an interesting mechanism for data security by an extension to strong typing called typestate checking. Each operation of a data type is assigned a pre-typestate and for each possible outcome a post-typestate. This allows checking across communication interfaces as well as a uniform method for handling exception outcomes. This mechanism, together with some other restrictions on branching, is used to verify program correctness without theorem proving.

---

[2] CSP is certainly an example of a programming technique but does not qualify, based on Hoare's original paper, as a modern language with a supported environment.

2.3. Graphical Techniques

Graphical techniques which employ states or control flow are useful but result in a combinatoric explosion of nodes and arcs when describing concurrent processes for all but trivial cases. These techniques are better suited for analysis of certain specific problems with a small number of states. They are not suited to specification of systems because of the combinatoric explosion.

2.3.1. Petri Nets

A Petri net [Agerwala 79], [Peterson 77] is an abstract formal model of information flow. This model is capable of modeling asynchronous and concurrent activities. Instead of the full details of the theory, two examples from the literature will be given. The first, by Agerwala (Figure 2-1) indicates the usefulness of a Petri net in modeling concurrency and conflict. The model in Figure 2-1 represents a single processor devoted to servicing two devices that are gathering data from the outside world. The cycle on the left of the figure represents device D_1 and the cycle on the right device D_2. Device D_1 obtains new data (firing of t_1) only when the previous data has been transmitted (token in p_1). Completion of this activity is signalled by a token in p_2. Under these conditions, if the processor is available it executes the service routine for I_1 and signals that the transmission is complete by placing a token in p_1. The whole cycle for I_1 can then repeat. The cycle for D_2 is quite similar. The second, used by Sorensen [Sorensen 78] (Figure 2-2), demonstrates the difficulty in partitioning Petri nets to reflect the structure of the problem. Sorensen shows his model where the interface between P1, P2, and P3 is clean, and independent processes or groups of processes can be isolated. This is often difficult in a Petri net model as shown in Figure 2-2 where the bus is represented by the place M.

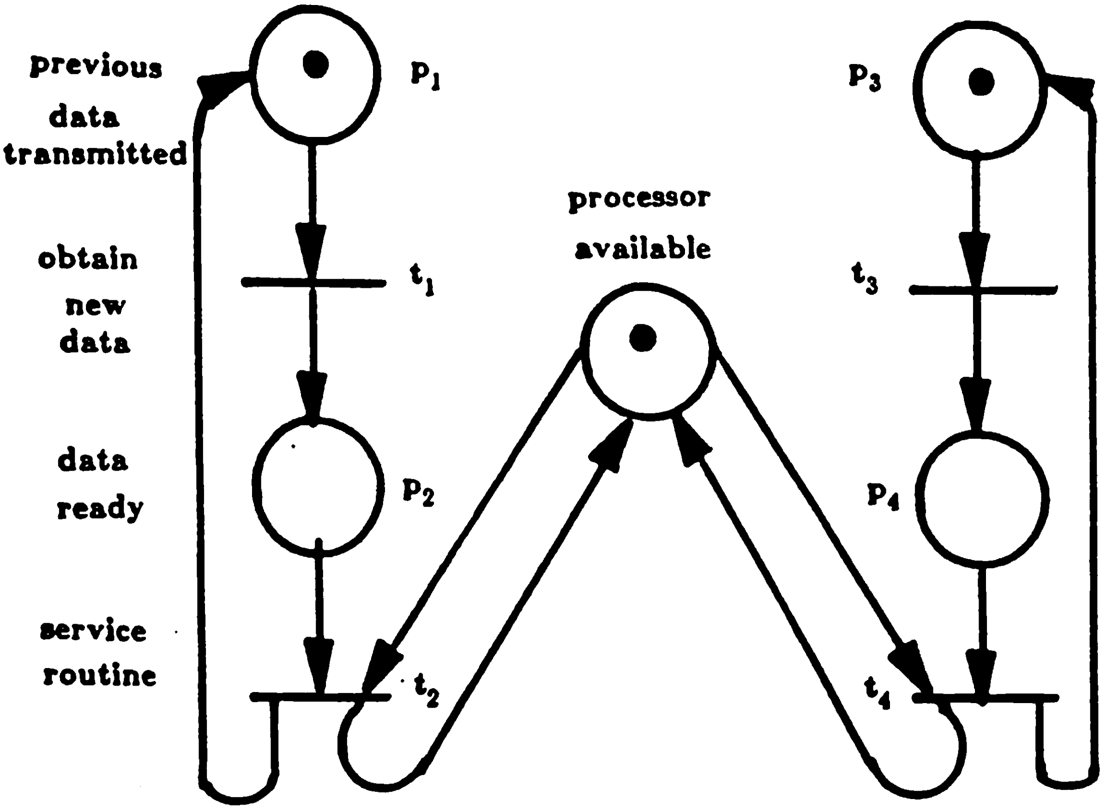


Figure 2-1: A Petri Net model of a processor servicing two devices.

2.3.2. UCLA Graphs---Graph Model of Behavior

A system is described in the UCLA Graph Model of Behavior by a data graph, a control graph and an interpretation. The Graph Model of Control (GMC) was shown to be equivalent to Petri nets [Gostelow 77], [Shapiro 83]. Since 1971, many extensions have been added to increase its modeling power; however, as pointed out by Shapiro most of these restrictions must be relaxed to verify system behavior using these graphs. Most of the current techniques for verifying the behavior of systems using

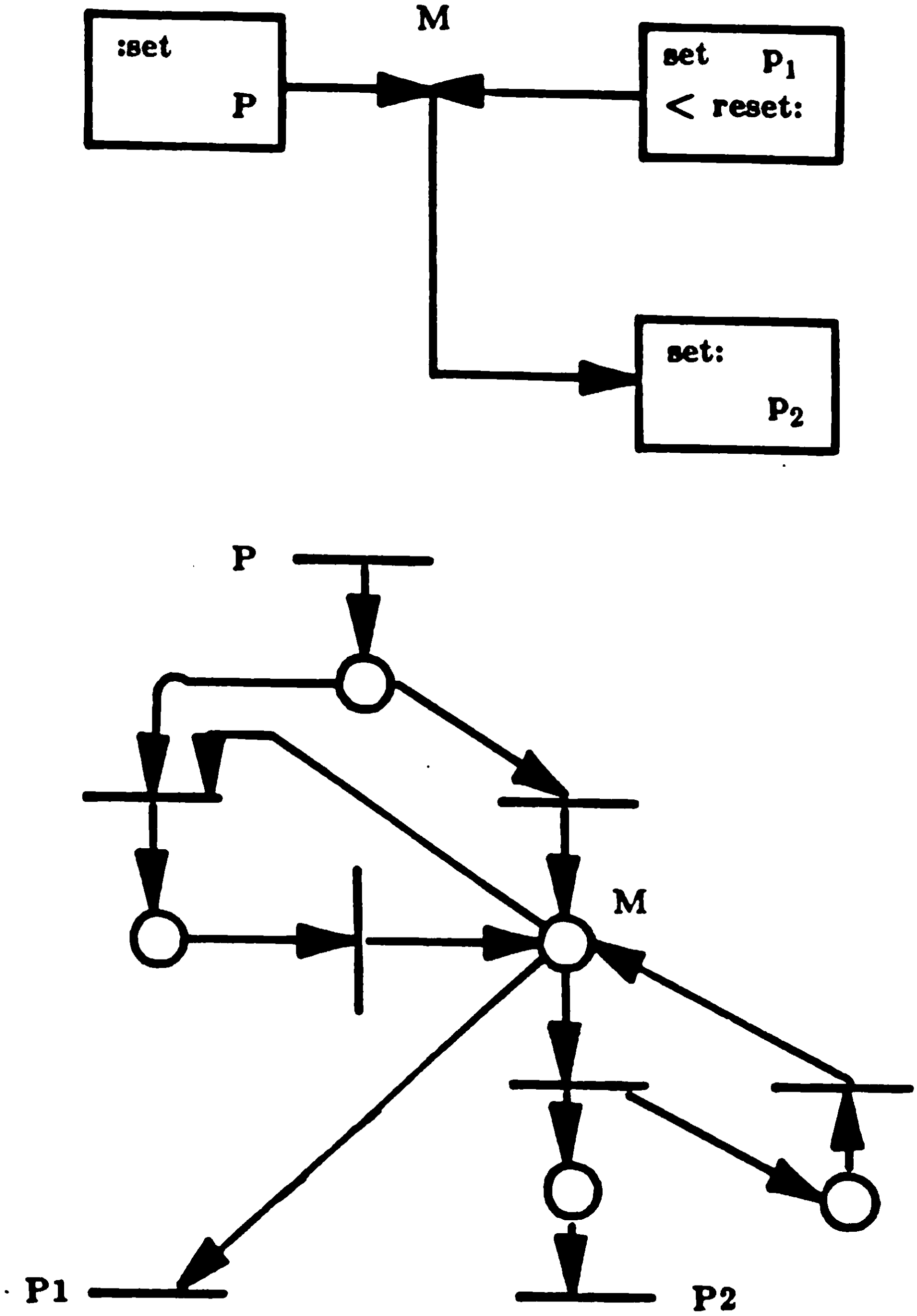


Figure 2-2: The use of a place in a Petri Net to model a common bus connecting the three processes.

this graphical technique involve exhaustive simulation and, therefore, offer little advantage over other representations which are based on or are extensions to Petri nets.

The GMB was developed for use in SARA.[3] The fact that SARA also depends heavily on building block models implies an implementation early in the design and hence a possible overspecification. The objective of the specification research proposed here is to follow the principle of least commitment and not to rely on implementation-oriented building blocks.

2.4. Mathematical Models

Three characteristics associated with mathematical models [Dijkstra 81] are relevant to the problem of specification. These characteristics are

- generality,
- precision, and
- provable properties.

Four mathematical models are explored in detail here: behavior expressions, predicate path expressions, temporal logic and the CCS (Calculus of Communicating Systems.

2.4.1. Behavior Expressions

Since Behavior Expressions (BEs) [McFarland 81] are an abstract model of behavior, we first introduce a less abstract behavioral representation, ISPB, which will be used as an aid in explaining BEs. ISPB is a subset of ISPS discussed in Section 2.2.2.1. The actions defined for ISPB are contained in Table 2-1. The atomic actions control the internal

---

[3] SARA (System ARchitect Apprentice) is a system developed at UCLA for computer-aided design of computer systems.

working of the machine by changing local variables and altering control flow. They also cause interactions with the external environment by reading and writing global variables. In ISPB a special naming convention is introduced to differentiate global or external variables from local variables. Lower case letters u,v and w, sometimes with primes and subscripts, stand for local variables. The lower case x is reserved for global or external variables. Programs in ISPB constitute a low-level behavioral description. All interaction with the environment is through the global variables. The local variables determine the state of the machine. The following definitions are quoted directly from McFarland.

Definition 2.1: An *event* θ in ISPB is a triplet of the form <read, x_i, c> or <write, x_i, c> or λ(the empty event) where x_i is a global variable and c is a constant value in the domain of x_i. An event is a fully interpreted interaction with the environment.

Definition 2.2: A history η is a sequence of events. It may be finite or infinite.

Definition 2.3: A behavior is a set of histories.

Definition 2.4: If M is a machine description in ISPB, Bh(M) is the behavior of M, *i.e.*, the set of histories generated by M.

Based on these four definitions, McFarland constructed BEs by augmenting regular expressions with predicates to show data dependencies. These predicates allow each *event* in an expression to be dependent on the past history of inputs to the program and the number of times any loops in which it is embedded have been *executed*. The latter dependency is made possible by assigning to each loop in the expression a *loop counter*, which is an integer variable different from any program variable. Since a machine can have a large set of histories, some of which are repetitive, it is

Action	Definition
$u \leftarrow e$	evaluate expression e and place in u
$x_i \leftarrow e$	evaluate expression e and place in x_i
$u \leftarrow x_i$	read x_i and place in u
leave f_k	exit procedure f_k and return to calling point
restart f_k	return to beginning of procedure f_k and continue execution
skip	skip to the next action
<u>Complex Actions</u>	
if b then A_1	if the Boolean expression b is true execute A_1
else A_2	otherwise execute A_2
$A_1;A_2$	the execution of actions A_1 and A_2 is order-independent
A_1 next A_2	complete action A_1 before beginning A_2
call f_k	call procedure f_k

Table 2-1: ISPB Actions.

convenient to use regular expressions [Shaw 80] to represent its behavior. An event schema E is a string of the form Λ, $R(x_i)$ or $W(x_i)$ (*i.e.* null, read or write) for $x_i \in \underline{x}$, the set of global variables. An atomic BE is a pair of the form E:P where E is an event schema and P is a predicate. An example of an atomic BE is

$$W(x_1):(x_1 = 0 \,\&\, x_2 = 1)$$

Its meaning is that "the value 0 is output to x_1 and the last value input at x_2 was 1."

Three functions designated T, L and R can be used to transform a precondition and an ISPB atomic action into an atomic BE. To apply these functions, we take an ISPB program and compute preconditions and postcondition for every action in the program, following basically the method of Floyd [Floyd 67] adapted to the peculiarities of ISPB. The precondition of an action is a logical formula which states what must be true of the internal variables, the external variables and the loop counters just before the action is executed. The postcondition is a similar formula describing the state of the system and its history just after the action has been executed. The function T, takes an action A and a precondition P for that action and produces a postcondition. The T function is shown in Table 2-2. Then these atomic BEs can be easily transformed into complex BEs which show sequencing, parallelism, choice or looping. This composition of BEs is facilitated by the fact that ISPB complex actions translate one-to-one into BE operators. A conditional statement translates into "+"(OR), a "next" to a "." (concatenation or sequence), a ";" to a "||" (order independence) and a procedure "call" to a loop.

The purpose of the other two functions L and R is to compute "leave" and "restart" conditions respectively. Behavior expressions are one

A	T[A]P
<u>Atomic Actions</u>	
$u \leftarrow e$	$\exists u'\ P <u \leftarrow u'> \wedge u = e <u \leftarrow u'>$
$x_i \leftarrow e$	P
$u \leftarrow x_i$	$\exists u', j'_i\ P<u \leftarrow u', j_i \leftarrow j'_i>$ $\wedge j_i = j'_i+1 \wedge u = x_i(j_i)$
leave f_k	false
restart f_k	false
skip	P
<u>Complex Actions</u>	
if b then A_1 else A_2	$T[A_1](P \wedge b) + T[A_2](P \wedge \sim b)$
$A_1;A_2$	$\exists \underline{w}'_1, \underline{w}'_2 (T[A_1]((P \wedge \underline{w}_1 = \underline{w}'_1)<\underline{w}_2 \leftarrow \underline{w}'_2>)$ $\wedge T[A_2]((P \wedge \underline{w}_2 = \underline{w}'_2)<\underline{w}_1 \leftarrow \underline{w}'_1>))$
A_1 next A_2	$T[A_2](T[A_1](P))$
call f_k	$\exists l\ \{L[A_k,f_k](P_k(P,l)) + T[A_k](P_k(P,l))\}$

Table 2-2: Definition of T, taken from [McFarland 81].

of the most abstract forms available for expressing a machine's behavior. However, they are limited to the description of a single process behavior and cannot support explicit description of reading and writing of variables across parallel branches, *i.e.*, cooperating processes with different clocks or asynchronous branching as described in Section 1.4.

2.4.2. Predicate Path Expressions

Predicate Path Expressions (PPE) [Andler 79] are a high-level synchronization construct that is used to specify synchronization as part of the type definition of a variable. The data abstraction will therefore contain the representation of that data type as well as definitions of all operators that can be applied to objects of that type. The PPE then specifies the allowable sequences of operations on an object of that type.

An example of a type definition follows:

The data representation: **message**

The operations: ***write*(buffer,message) and**
***read*(buffer) --> message**

The synchronization: **path (*write.read*)***

The dot "." expresses sequencing, *i.e.* write has to be performed before read. The Kleene star ("*") expresses repetition of the entire sequence, *i.e.*, we can perform another *write* once a *read* operation has consumed the previous message.

Note that since the definition occurs where the shared object is defined and not where it is used this enhances the readability of the description much like the procedure abstraction mechanism. Also by adding predicates to path expressions, Andler has avoided having to introduce a large number of special purpose operators [Brinch Hansen 73] for the different types of synchronization problems.

Another example taken from Andler's thesis is the shared bounded stack. The PPE is:

```
type stack [int max] =
     def items = term(push) - term(pop)
     path (push[items < max] | (pop|top)[items > 0])*
```

where the | represents selection. The path expression describes the mutual exclusion of the *push*, *pop*, and *top* operators. It also specifies that *push* can only be applied when *items* (the number of elements currently in the stack) is less than the maximum size *max*, and *pop* and *top* can only be applied when there are elements in the stack, *i.e.*, *items* is greater than zero.

Although PPEs are limited to the class of Communicating Sequential Processes, the close relationship between the concepts of history and the different nature of the predicates indicates that a significant increase in expressive power might be achieved by combining this representation with that of BEs.

2.4.3. Temporal Logic

Temporal logic [Rescher 71] is an extension of standard logic to time-related propositions. Temporal logic has been used by several researchers [Pnueli 77], [Hailpern 80], [Bochmann 82], [Lamport 83], [Moszkowski 83], [Fujita 85] to reason about temporal issues associated with programs, network protocols, and most recently hardware. The general level of abstraction presented by Bochmann, Moszkowski and Fujita was at a level where the individual states of the device had to be identified and much of the reasoning had to be done in terms of signals and their levels. Also the description of the behavior mixes structural, data flow and sequencing and timing information in a way which makes it difficult to reason about any one of these individually. Other work by Schwartz *et al.* [Schwartz 83] which could be applied to this research has introduced intervals and used temporal logic to reason about these intervals

2.4.4. Representation of Temporal Information

In addition, to the classic view of temporal logic described in the previous section, other work on representing temporal information has been done in the area of artificial intelligence [Bolour 82]. The work of Allen [Allen 83] is closest to the research done in extending the DDS timing and sequencing representation. The basic similarities are the notion of the temporal interval as a primitive and the characterization of the relationships between temporal intervals in a hierarchical manner using constraint propagation techniques. Our research differs in that we have added a different notion of points and have extended the semantics of the relationships to reflect causality. This is discussed in detail in Chapter 3.

2.4.5. Calculus of Communicating Systems

Milner's original work [Milner 80] describes a mathematical semantics for concurrent computation and communication. The main concern of this work is proving the semantics of the model and that the communication operations provided by the calculus form an *algebra* with the right properties. The original work also focused on asynchronous behavior and did not describe synchronous behavior. In a later work [Milner 83], a more general theory of synchronous behavior was developed and a asynchronous behavior is treated as a subclass of the synchronous calculus. The primary primitive concepts that give the Calculus of Communicating Systems (CCS) its modeling power are value-transmission and message passing, the notion of a port, and rules governing how processes connect and interact. As with other models discussed in this chapter, CCS's primary deficiency is an inability to deal with detailed timing considerations. Lamport [Lamport 83] also questions the capability of this type of approach to support hierarchical specification.

CCS has been used successfully by Milne as a basis for CIRCAL (CIRCUIT CALCULUS) [Milne 83] and Gordon who extended the CCS model to handle register transfer systems [Gordon 81a], [Gordon 81b]. CIRCAL is suitable for low-level elements such as nMOS and CMOS transistors, inverters, gates and storage elements. Both of these models reflect the structure of the hardware at each level of description, making it difficult to construct an abstract specification without implementation bias. Furthermore, Gordon's model only supports fully synchronous designs with a single clock.

2.5. Informal Techniques

System specification is currently done almost exclusively using informal techniques.

2.5.1. Timing Diagrams

Timing diagrams are the primary representation used by designers in specifying production hardware systems. They represent a register-transfer, logic and/or circuit level of design. Since the semantics of these diagrams are not completely defined but certain *ad hoc* notions are fairly common, a large amount of information is usually represented in one of these diagrams. It is likely that half the information is contained in notes associated with various events and edges and even the connotations associated with the signal names.

Some of the ambiguous interpretations which might be associated with an interval between two transitions on an uninterpreted graph are: minimum delay, maximum delay, measured delay or causality (*e.g.*, one edge may or may not have caused the other transition).

However, due to their ubiquity and popularity among designers, timing diagrams may represent a useful graphical representation to display information from a more formal internal representation at the user interface [Booth 81]. If so, the semantics of timing diagrams must be much more clearly defined.

2.5.2. The Previous Design Data Structure

The previous DDS has been characterized as informal for the purpose of this discussion because the semantics, especially those of the timing and sequencing subspace, had not been fully defined when this research was initiated. Only a brief overview of the DDS will be presented here, since the semantics of the Design Data Structure (DDS) are described in detail in Chapter 3. The DDS is a unified representation of design data. It has been designed to support and facilitate the synthesis of digital hardware systems. It is composed of four subspaces, each of which may be divided hierarchically as desired to decompose or compose abstractions or implementations in the design spaces. The subspaces are

1. **Data Flow**: which covers data dependencies and functional definitions. It is represented as a bipartite acyclic graph where one type of node represents the operations and the other type of node represents the values. The arcs which connect these nodes indicate the sources and sinks of the values. These graphs may be viewed as a single assignment programming language.

2. **Timing and Sequencing**: which covers timing, sequence of events and conditional branching. It is represented by a directed acyclic graph, which consists of nodes corresponding to

events, and arcs which represent intervals and connect these nodes. To capture as much semantic information about the design as possible, four types of arcs and seven types of nodes are used to model various aspects of timing and control (for example, concurrency, choice and constraints).

3. **Structural**: which covers the logical decomposition of a circuit. This subspace is similar to a schematic or block diagram. It consist of modules which are interconnected by carriers.
4. **Physical**: which covers the physical hierarchy of components and the physical properties of these components. In this subspace there are two primitive object types: blocks and nets.

The relationships between these various spaces are made explicit by means of bindings. These bindings and the information in the four subspaces are believed to fully characterize the design. The complete syntax is presented in Knapp and Parker's report and some of the important semantic notions are also operationally defined. In this research, we are primarily concerned only with the "behavioral" subspaces, the Data Flow subspace (DFss) and the Timing and Sequencing subspace (TSss). Though the semantics are not formally defined, the DDS satisfies the other criteria required for a representation of digital design information.

2.5.3. Natural Language Processing

In this section, we will review only the closely related work among recent research in the field of natural language processing.

Previous work done on processing natural language specifications has been concerned primarily with software systems [Balzer 85], [Mander 79], programs [Abbott 83], [Ginsparg 77] and data types [Comer 79]. This work falls into two categories. The first is characterized by virtually unrestricted application domains and therefore requires enormous vocabularies and the ability to deal with tremendous variability in the input. The second covers a very limited domain; namely, the manipulation of the objects which are created from the specification, e.g. CREATE A STACK, DELETE A SET, etc. Also, it should be noted that Mander was only concerned with syntactic analysis.

One prior endeavor involved the application of natural language processing as an input to a design system for digital electronics [Grinberg 80], but this work actually focused on the construction of a circuit given predefined components and was focused on implementation rather than specification. Furthermore, it used certain hyphenated verb forms, e.g. IS-CAPTURED-IN, and noun phrases like NUMBER-OF-WORDS to aid in the processing making it more like an application-oriented **programming** language.

Other recent works, like the UNIX Consultant (UC), [Wilensky 84], and **CLEOPATRA** (Comfortable Linguistic Environment that Obstensibly Permits Arbitrary Textual Requests and Assertions) (Samad86), answer questions concerning a given body of knowledge, the former the UNIX operating system, the latter the results of a digital simulation.

The research described here differs from **UC** and **CLEOPATRA** in that it is creating an entity, *i.e.*, a formal, neutral representation of the behavior being specified. To create this representation, semantic knowledge about system behavior has been encoded in the parser's knowledge base.

The work by Fink, Sigmon and Biermann [Fink 85], on a limited natural language control of a machine in a task-oriented situation should be mentioned. They concluded that "...often-mentioned concerns related to the lack of precision of natural language were not a problem in the domain of our experiments."

2.6. Deficiencies in Existing Techniques

The most common deficiences associated with existing techniques for specifying digital designs are

1. lack of a formal abstract model of behavior,
2. inability to represent asynchronous behavior of concurrent processes,
3. unsuitability for large problems,
4. lack of constructs for specifying performance – especially timing,
5. lack of high-level specification constructs,
6. inability to produce comprehensible descriptions, and
7. lack of tools for constructing the specifications

Even though considerable work has been done in the many areas of related research described in this chapter, no language, method or technique has explicitly and systematically addressed these deficiencies. The approach taken in this research has been based on a set of requirements that explicitly address all of these deficiencies. In addition, this research tried to incorporate and build on the results of this related research wherever appropriate. Specifically, the PHRAN parser, a component of the Unix Consultant was used to construct the prototype system, the DDS was selected as the underlying representation for system behavior and Allen's work on temporal representation was used as a basis for extending the timing and sequencing model of the DDS.

Chapter 3

The Design Data Structure: A Specification Tool

3.1. Introduction

This chapter will discuss the use of the USC Design Data Structure (DDS) as an abstract representation for the specification of digital system behavior. The USC DDS was introduced in 1983 [Knapp 83], and is further described in other USC publications [Knapp 84, Knapp 85]. However, previously published material has concentrated on the overall DDS concepts and usage. Here, we focus on the semantics of DDS constructs, as required for system specifications. Since a specification of system behavior involves the *what to do* as contrasted with the *how to do it*, this discussion will focus primarily on only two of the DDS subspaces; namely, the data flow subspace (DFss) and the timing and sequencing subspace (TSss). The other two subspaces, the Structural subspace (Sss) and the Physical subspace (Pss) will be introduced as required to handle aspects of the specification that are not expressed in the DFss and TSss. In addition, the Structural and Physical subspaces may be used in specifying a system if there is a strong desire or need to constrain the implementation details. The DDS also includes several types of relations among the various subspaces. These relations are termed *bindings* and will be defined when they are introduced. Finally, additional information or ancillary data may occur in a specification and not be represented or representable in the DDS. The topic of ancillary data will be covered in Chapter 4.

All the subspaces in the DDS are hierarchical; however, the semantics of the DFss and TSss will first be described as a one-level space and any effects associated with hierarchy will be introduced after the basic constructs have been defined. While there is no certainty that this extension to the DDS makes the behavior complete, it is sufficient to capture the semantics of ISPS and SLIDE.

3.2. The Data Flow Subspace

The DFss may be formally described by a bipartite directed acyclic graph (bi-DAG). The two types of nodes are **data flow operation nodes** and **data flow value nodes**. The nodes are connected by **data link arcs**, which associate the data flow operations with the values. The bi-dag allows this representation to serve as a single-assignment data flow *programming* language [Tesler 68] and avoids the confusion associated with the naming of values. (Others, notably Dennis [Dennis 74], introduced tokens and *splitter* nodes to solve this problem.) The values may be treated symbolically; that is, a value may be referenced as A or *foo*. Such a symbol may be associated with a numeric value, for example, four or π or some particular sequence of bits. This is in contrast to the common notion of a variable that may have more than one value associated with it during its lifetime. Instead, this *temporal behavior* is modeled by *binding* the values to intervals in the TSss and also *binding* to carriers in the Structural subspace. Since a value can be bound to a carrier during an interval of time, a variable can be represented in the DDS by a sequence of values, each bound to a different interval but a single carrier. An example of the DFss is shown in Figure 3-1. The primary information represented in the DFss is the number and type of operations required, the number and type of inputs and outputs to the various operations, and the data dependence associated with the operations.

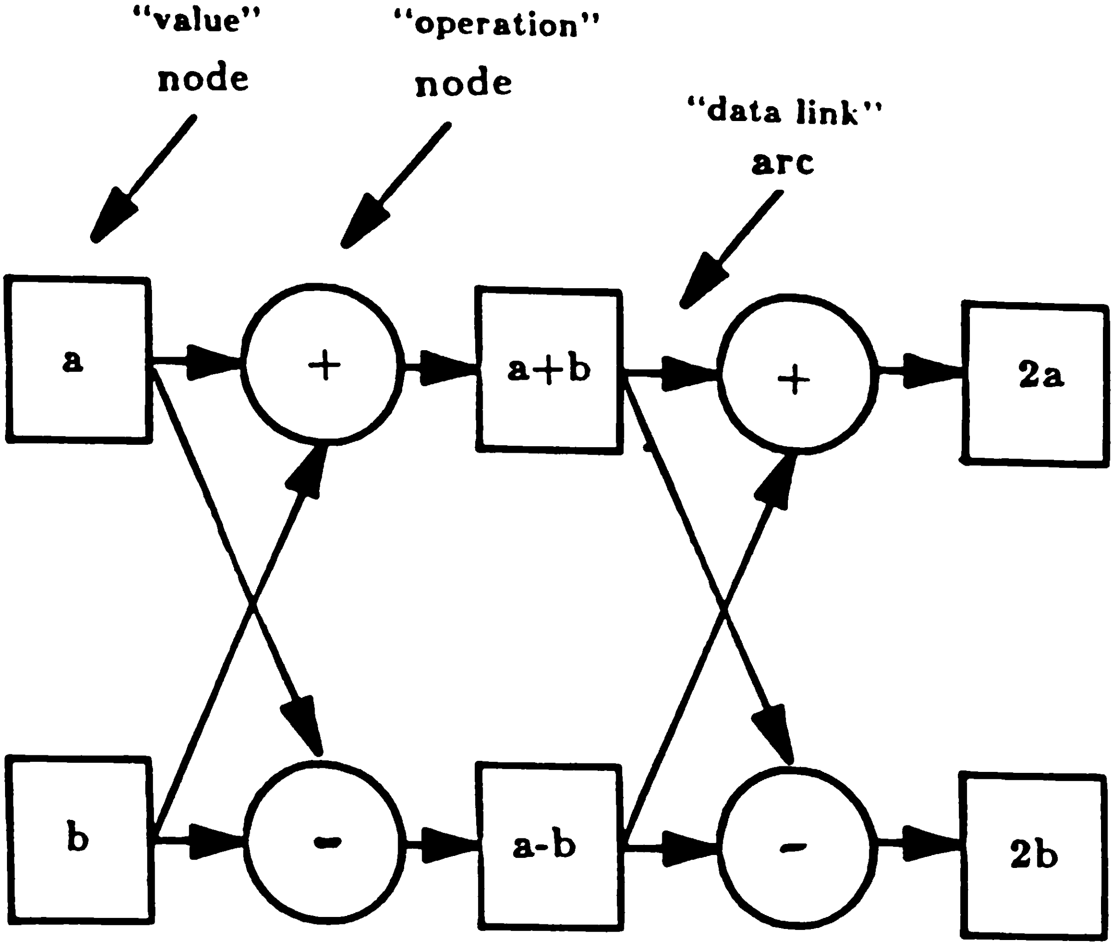


Figure 3-1: A Data Flow Subspace Example.
This example, called *crisscross* is taken from [Hafer 81].

The Dfss operations may be highly abstract (*e.g.*, Kalman filter) and defined at a later time in terms of the primitive DFss operations for which truth tables are defined in the library (the hierarchy permits this abstraction/decomposition).

Examples of primitive operations are

- the Boolean operations not, and, or, nor, nand, and xor;
- the relational operations $>$, $<$, $=$, $\geq$, $\leq$,
- the arithmetic operations add, and subtract; and
- the control operations *select* and *distribute*.

The select and distribute operations will be described later.

The basic representation has been augmented with subscripts for both the data flow operations and the values. Two types of subscripts are associated with the values: The first is a subscript for describing values that are actually composed of groups of bits; *e.g.*, an 8-bit wide input for an 8-bit adder would be written a[7..0] where a[7] would be the leftmost bit in a left-to-right representation and a[0] would be the rightmost bit. Note the use of square brackets to signify a spatial arrangement of values.

A second type of subscript indicates a temporal sequence of values; *e.g.*, a stream of bits would be written as b(0..N) where b(0) would precede bit b(N) in a sequence. These two types of subscripts may be transformed by providing storage, serial-to-parallel converters or parallel-to-serial converters. Both subscripts may be used together. No syntactic convention or semantic significance is associated with the order of the type of subscripts, *e.g.*, a(5..9)[3..0] and a[3..0](5..9) are equivalent.

3.3. Timing and Sequencing Subspace

The Timing and Sequencing subspace (TSss) is formally represented by a directed acyclic graph (DAG) model. There are four types of arcs in this model. The four types are based on the semantic use of the arcs in representing timing and sequencing information.

There are also seven types of nodes in this model. First, the types of arcs will be defined. Next, the various types of nodes will be described. Finally, the various combinations of nodes and arcs allowed in this model will be discussed.

3.3.1. TSss arc types

The four types of arcs are sigma (σ) arcs, theta arcs (θ), chi (χ) arcs, and delta (δ) arcs.

A **sigma arc** represents an interval of time (or range [Knapp 83]) in the TSss. A sigma arc may also be viewed as a sequence of events or points. However, since a point has no actual dimension (like a geometrical point on a line), the points serve only as labels used for reference to specific events. The duration or length of the interval is associated with the arcs joining the nodes. Since the ultimate objective is a physically realizable implementation, one cannot bind an operation or a value to a node or point; bindings are only permitted to arcs. Finally, sigma arcs may be assigned a specific length that indicates a particular amount of time (units are established as required by the design) as shown in Figure 3-2. The ends of the sigma interval in this figure are referenced as π_1 and π_2.[4] Note, this length is defined in terms of a value and a relation. The relation may be any of the following: $>$, $<$, $=$, $\geq$, $\leq$. There is no restriction that the length of a TSss arc be positive; however, time proceeds in only one direction and negative lengths can be transformed to positive lengths by reversing the direction of the arc.

---

[4] In this representation a pi point is a simple event and will be explained in Section 3.3.2.

A **theta arc** represents a temporal constraint. For example, if the beginning of one interval is specified to occur at 100ns after another interval ends, a theta arc is used to represent this information. An example of this is shown in Figure 3-3.

A **chi arc** represents a causal relationship. An example of this type of arc is where the end of interval σ_1 is causally related to the beginning of σ_2, *i.e.*, σ_1 ending causes σ_2 to start (shown in Figure 3-4).

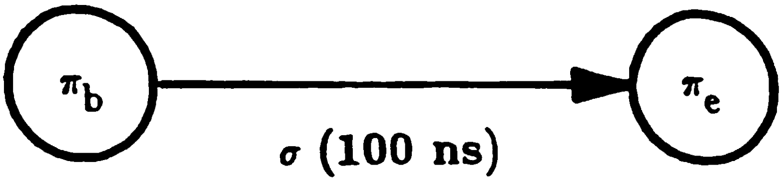


Figure 3-2: A sigma arc in the TSss and it length in nanoseconds.

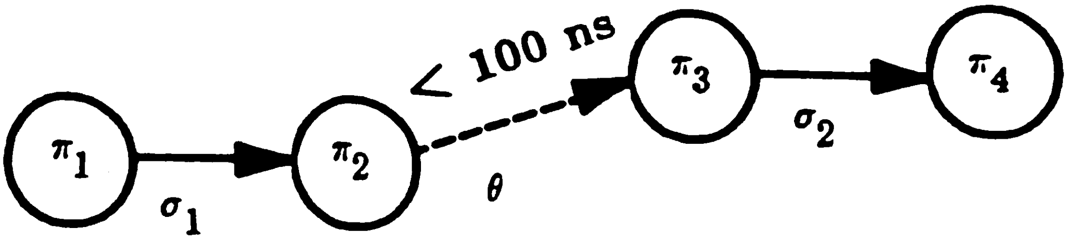


Figure 3-3: A theta arc used to specify that one interval begins 100 ns after the end of the other.

A **delta arc** represents *inertial delay* [Breuer 76].[5] In the research presented here, a simplified model is used that lumps the various delays associated with a physical component. The lumped delay, δ_1, as shown in

---

[5] Inertial delay is not equal to propagation delay.

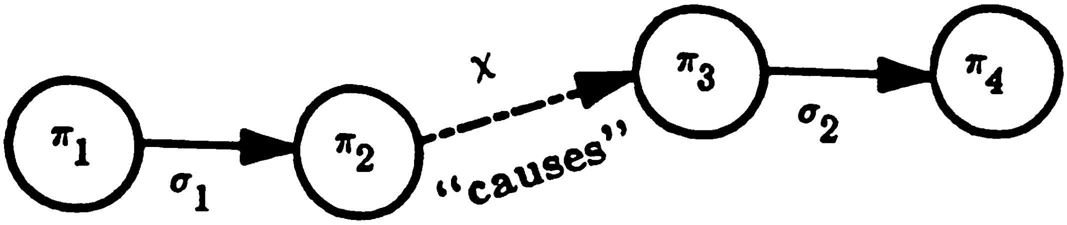


Figure 3-4: A chi arc representing a causal relationship.

Fig 3-5 is associated with the interval, σ_{in} that begins with the arrival of the last input and ends with the beginning of the output being valid, σ_{out}. The end of the interval, σ_{in} is constrained by the arc labelled, $\tau > 0$ to precede the beginning of the interval labelled δ_1. This constraint simply stated is that the input value must not end before the operation begins. The analogy in a physical implementation would be to measure the output after the signal was removed and the input was floating.

Note that the DDS can be used to construct a detailed timing model at the transistor level, if required. Obviously, such a model is not required for system specifications.

3.3.1.1. Notational convention

The symbol **phi** (ϕ) is so often associated with a clock in digital designs that this symbol will be reserved for that purpose; semantically, a clock interval is not fundamentally different from an arbitrary regular repetition of **sigma** arcs.

Also, the symbol **tau** (τ) will be used to refer to the fundamental unit of time chosen as a quantum unit for a particular interpretation. No interval of smaller duration than this may be distinguished under that interpretation.

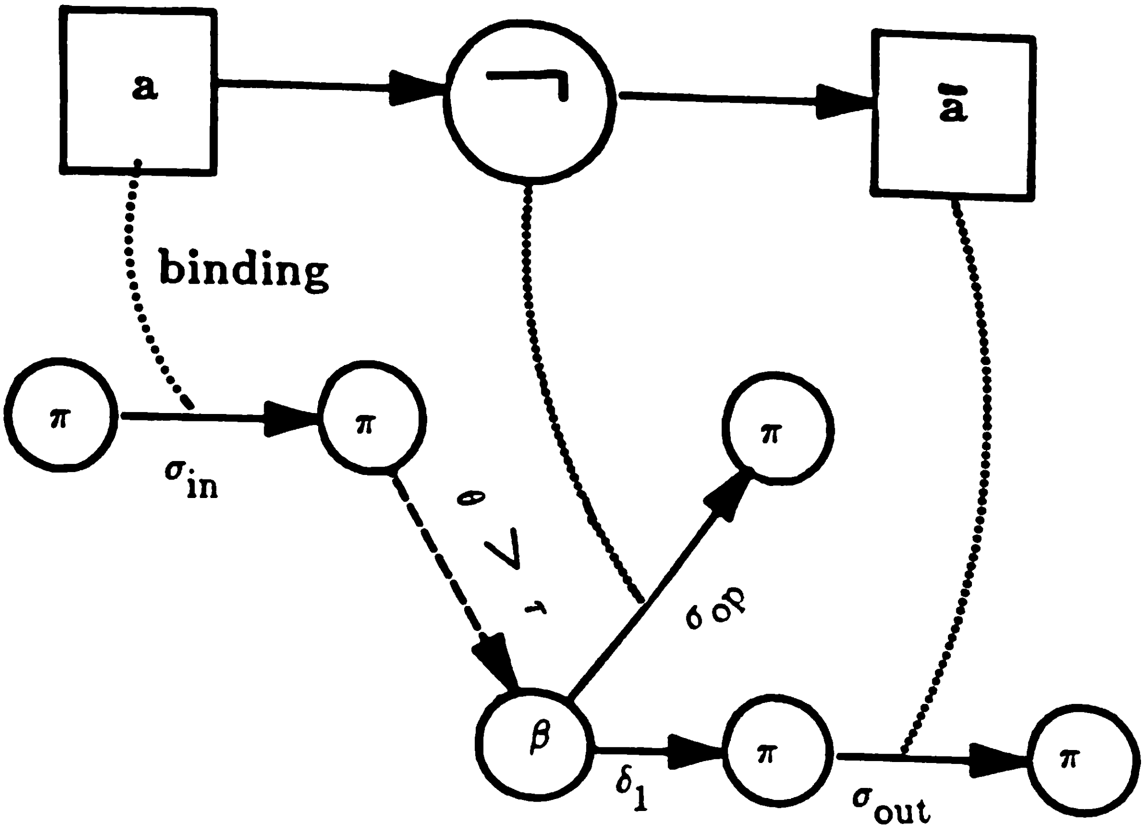


Figure 3-5: A DDS representation showing the use of a delta arc to model the delay associated with an implementation.

3.3.2. TSss node types

There are seven types of nodes in the TSss: pi (π) nodes, beta (β) nodes, gamma (γ) nodes, mu (μ) nodes, rho (ρ) nodes, alpha (α) nodes, and omega (ω) nodes. The first type is a simple node that may *join* two arcs, providing a label for the *meets*[6] relationship [Allen 83] or providing a label for an event. This is a pi node or *point*. Some examples are shown in Figure 3-6.

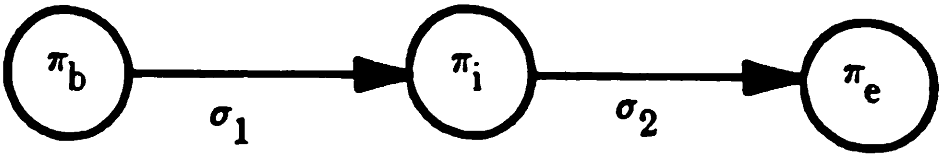


Figure 3-6: Three pi points joined by two sigma arcs.

The location of π_i is within the interval between π_b and π_e but is not further specified.

The remaining six types of nodes are only useful to establish the temporal relationship between three or more arcs. First, the types of nodes will be described with respect to sigma arcs only. The various combinations of nodes and arcs are defined in Section 3.3.4.

A **beta node** represents a point at which the end of one interval is synchronously associated with the beginning of two or more other intervals. This may also be referred to as an *and fork* point [Conway 63] or a *cobegin*

---

[6] Meets is one of the thirteen unambiguous relationships that Allen defines between any two intervals in time. It is a graphical relation in which the end of one interval abuts the beginning of the following interval.

[Dijkstra 68]. The two branches *begin* together but no additional information is implied in Figure 3-7 (Note: Allen [Allen 83] uses the label *starts*, which seems to imply some causality; the model described here separates the causal information by using the chi arc construct.)

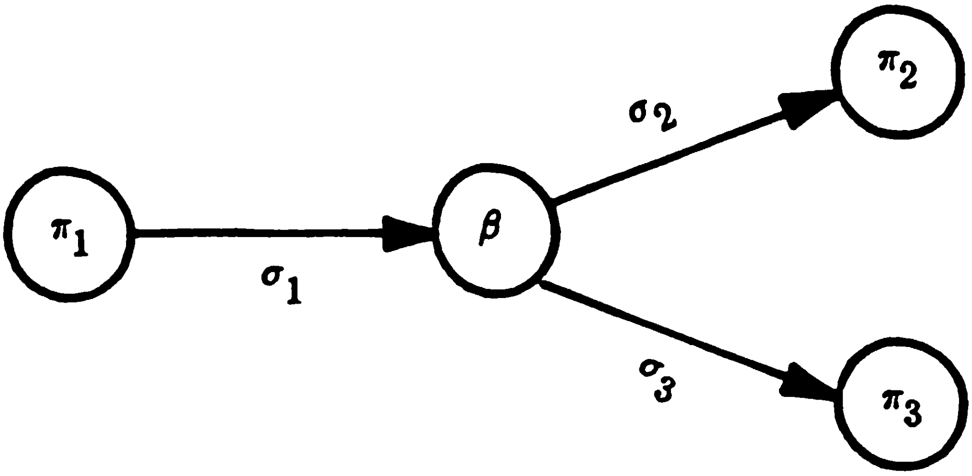


Figure 3-7: A two branch **and fork** example.

A **gamma node** represents a point at which the end of one interval is associated with one of a set of subsequent intervals, thereby representing an n-way branch. Each branch exiting from this node is an exclusive selection. The choice of branch is based on the value of a predicate that is attached to each arc emanating from the gamma node. The predicates will be discussed further in the next section with respect to their use in describing asynchrony and in the section on DDS canonical templates. A gamma node, two branch example is depicted in Figure 3-8.

A **mu node** represents an *and join* point, *i.e.*, the termination of two parallel branches. This node is analogous to a *coend* [Dijkstra 68]; appropriate delay is inserted in either branch to insure concurrent termination. An example of an **coend** is depicted in Figure 3-9.

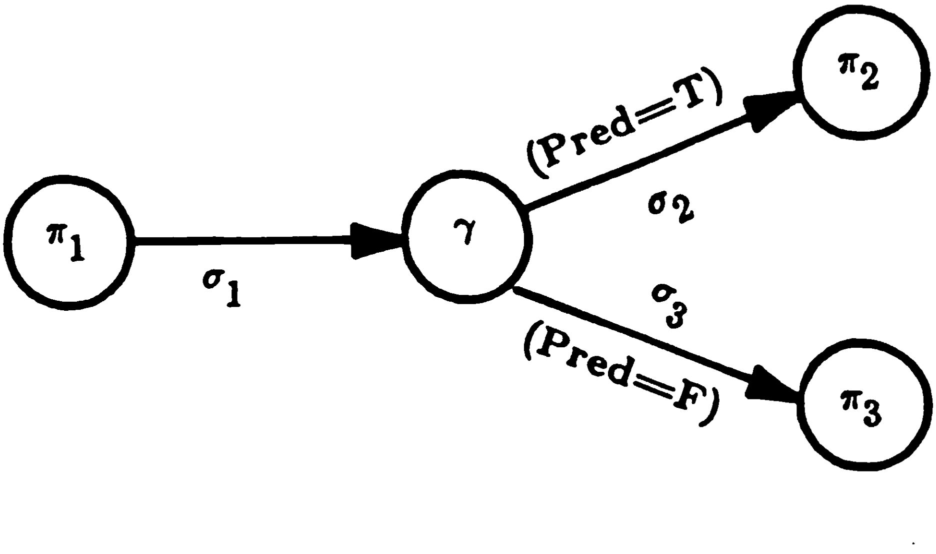


Figure 3-8: A two branch **or fork** example.

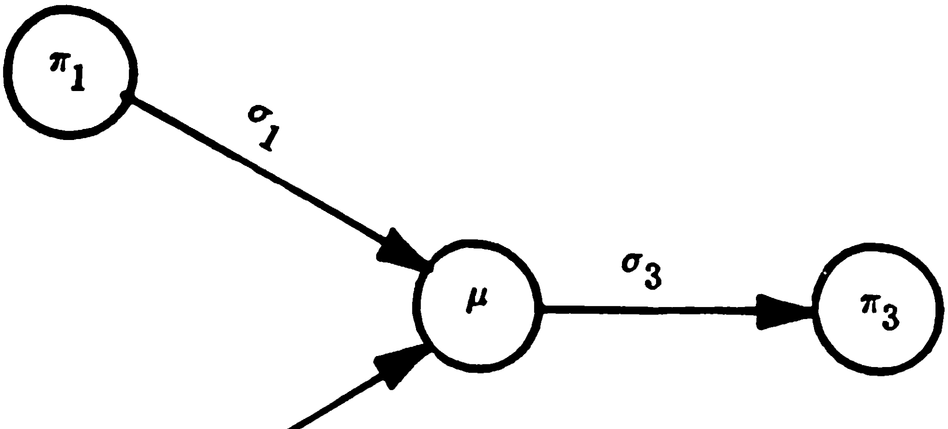


Figure 3-9: An example of **coend** in the TSss.

A **rho node** represents an *exclusive-or join* point. The arcs that terminate at this point represent all possible branches that could be the predecessor of the arc emanating from the join. Only one branch (arc) will actually be active in a properly specified behavior. An example of a rho node joining three sigma arcs is shown in Figure 3-10.

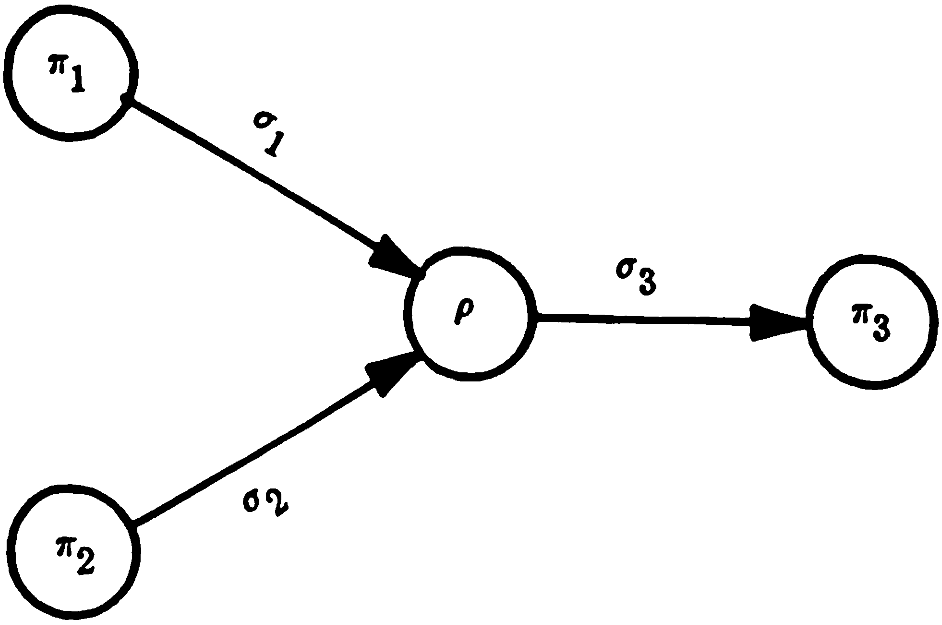


Figure 3-10: An example of **xor join** in the TSss.

Alpha nodes and **omega nodes** occur in pairs and will be described together. An **alpha node** represents the beginning of a repetitive interval or loop. The arc or sequence of arcs that emanates from this point will eventually terminate in an **omega node** that represents the *normal* termination of the repetitive interval. The basic concept is shown in Figure 3-11 where the details of the TSss loop body are shown schematically. The alpha node and omega node are given symbolic subscripts. These subscripts are used in distinguishing values and operations in different iterations of the loop. When values are bound to a loop in the TSss, a correspondence between the value subscripts that are in parentheses () and the subscript of the loop is established. In effect, this loop could be considered to be *unrolled* in the DDS and is simply a sequence of subgraphs delimited by subscripted alpha and omega nodes as indicated in Figure 3-12. Unfixed loops, *i.e.*, those loops with an unknown number of repetitions and infinite loops cannot be unrolled. Also, since the arcs inside the loop are subscripted there may be a different length of time associated with every execution of the loop.

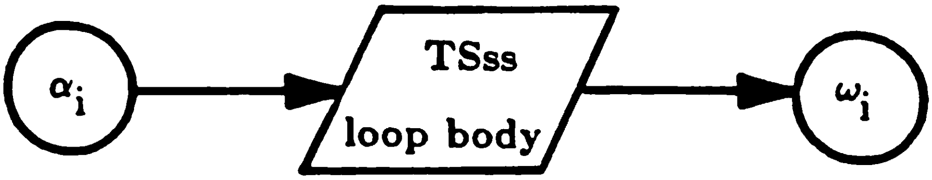


Figure 3-11: An iterative loop in the TSss.

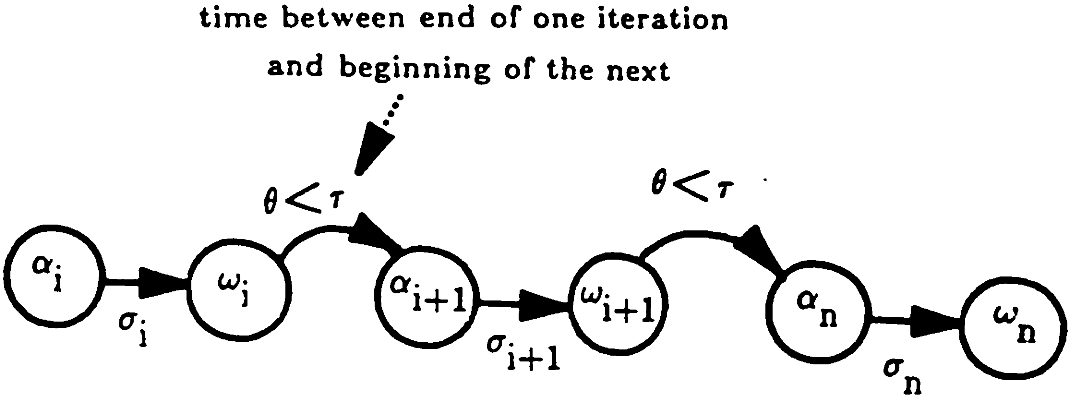


Figure 3-12: An iterative loop *unrolled.*

3.3.3. TSss predicates

The two types of predicates attached to timing arcs are

- **synchronous** predicates, and
- **asynchronous** predicates.

A synchronous predicate is attached to each of the arcs emanating from a gamma node. This predicate indicates the branch to be chosen at the time of *execution.* This corresponds to the familiar *if-then* conditional statement of most programming languages.

For example, the statement

```
IF (foo = True) THEN bar ELSE baz;
```

indicates that the action associated with σ_{bar} will occur only if **foo** has the value true when this statement is encountered; otherwise, the action associated with σ_{baz} occurs. In other words, σ_{baz} is associated with NOT-**foo** (**foo** = False). An example of this in the TSss is shown in Figure 3-13.

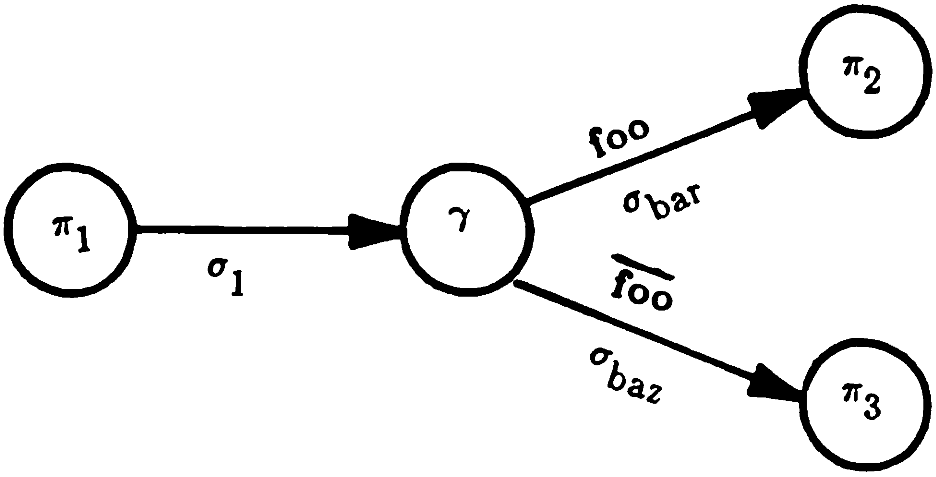


Figure 3-13: A gamma node representing a synchronous predicate. Note, this predicate must be valid during a time interval which occurs at or before the gamma node for this construct to represent *correct* behavior. Undefined predicates must be handled explicitly by the specifier of the behavior by including an appropriate branch for the undefined predicate.

Each predicate is a Boolean expression. An n-way branch from the gamma point would create a control structure like a CASE statement (Since one branch and only one branch may be selected at a gamma node, the OTHERWISE condition must be explicitly modeled).

The synchrony described by this first type of predicate is with respect to the gamma point. The gamma point may or may not be precisely fixed in time with respect to other components of the behavior. However, this type of predicate will frequently be defined in terms of indexed values which will be associated with alpha-omega loops; therefore, its temporal partial order will be established with respect to the loop.

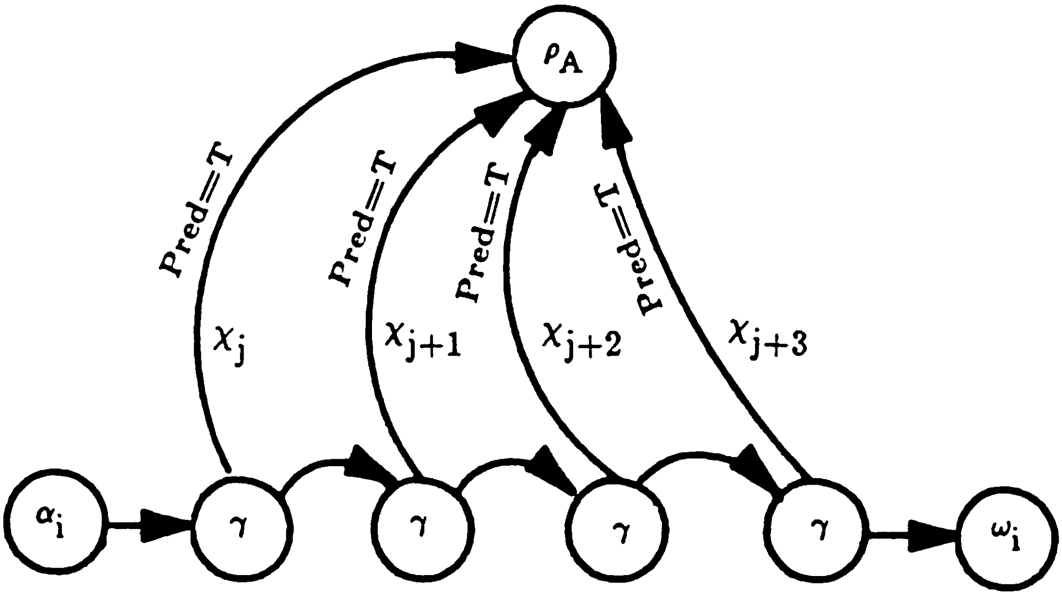
An asynchronous predicate is composed of an expression with a Boolean value that defines the conditions under which an asynchronous action will occur. In addition, the binding between the asynchronous predicate and an interval and the binding between the asynchronous predicate and a carrier must be specified and the destination point in time that defines the start of the subsequent behavior must also be specified. The asynchronous predicate makes use of the basic gamma node and its synchronous predicate and the property of nodes that makes them like dimensionless points. Therefore, a sigma arc with attached asynchronous predicate may be modeled as consisting of an infinite number of gamma nodes. By assigning the same predicate[7] to each of these gamma nodes and by terminating the one arc emanating from each gamma node at a single rho node and the other at the next gamma node, a model of asynchrony can be constructed as shown in Figure 3-14. Whenever the predicate becomes true the alternate branch at the particular gamma point is taken.

There are three additional constraints that are placed on this model: First, the arcs emanating from each gamma node and terminating on the rho node must be chi arcs (*i.e.* causal arcs). This constraint is imposed since the condition at the gamma node is causing a different sequence to be

---

[7] Though the basic behavior associated with the predicate with respect to the gamma node is the same as in the synchronous case the predicate is defined quite differently. This is explained in the remainder of this section.

followed. Secondly, a carrier from the Sss must be introduced to provide a mechanism for the predicate to change from false to true within the arc for which the predicate is defined (Since a value in the DFss has a single-assignment, there is no mechanism for *changing* a value--even if the value is symbolic. Thus it is the structural **carrier** which changes values). Finally, unlike the synchronous predicate, the asynchronous predicate is based on the bindings between the desired value in the DFss, the sigma arc in the TSss, and the carrier in the Sss. When the value which makes the predicate true is the *active* value on the carrier, the alternate branch is taken.



$\{B(\text{carrier, pred}=T, \sigma_1); \rho_A\}$

Figure 3-14: An example of asynchronous behavior in the TSss.

For notational convenience, the gamma nodes and chi arcs are not drawn but are implied by the presence of an asynchronous predicate associated

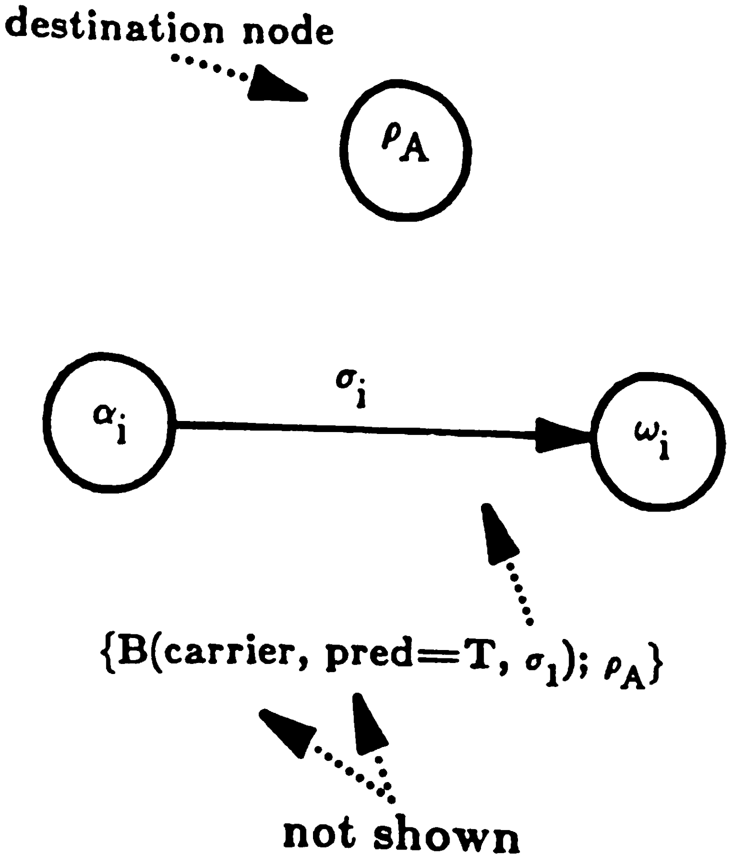


Figure 3-15: An example of asynchronous behavior using the abbreviated notation.

with a sigma arc. The form of the predicate is shown in Figure 3-15, where B signifies the binding relation from value to carrier to sigma arc and the destination node is the termination point of all the chi arcs.

This is the first example of a gamma node used with more than one type of arc. The incoming arc is generally a sigma arc and the outgoing arcs are of type sigma and type chi. The other arc combinations of type sigma-sigma and type sigma-theta may occur in the DDS, but are considered invalid combinations when used with an asynchronous predicate because they do not capture causality.

3.3.4. TSss arc/node combinations

In the following section, only sigma, chi and theta arcs will be discussed. Phi arcs are considered as special cases of sigma arcs and therefore possess the same semantics in combination with other arcs. The semantics of the remaining arc type, delta arcs, can be defined as a subset of the sigma arc cases associated only with physical implementation. In order to combine three or more arcs at a node, the node type must be permitted to have degree three or higher. There are four distinct node types that may have degree three or higher: gamma nodes, beta nodes, rho nodes and mu nodes.

First, the seventy-two cases associated with all combinations of three arcs and one node will be described, then the obvious extensions to n-way combinations will be noted. In general, the behavior of concurrent asynchronous digital systems can be composed using the seventy-two cases and appropriate constraints. The completeness of these seventy-two cases has not been verified as a part of this research; however, no description of system timing and sequencing has required additional constructs.

A notation will be introduced to allow a tabular presentation of the Arc/Node combinations.

$$\sigma_1\beta\sigma_2\sigma_3$$

This notation is equivalent to the graphical notation in Figure 3-7.

Having described the primitive object types in the TSss and DFss, interspace bindings, node semantics, control operations in the DFss, hierarchies and inheritance will be introduced and used for some examples of behavioral representation in the DDS.

InNode Out1Out2	Description
$\sigma_1\beta\ \sigma_2\sigma_3$	σ_1 terminates/meets σ_2 and σ_3 which both initiate concurently - cobegin
$\sigma_1\beta\ \sigma_2\chi_1$	σ_1 terminates/meets σ_2 and χ_1 which both initiate concurrently with a causal action propagated by χ_1 - UNIX(TM) -like "fork"
$\sigma_1\beta\ \sigma_2\theta_1$	σ_1 terminates/meets σ_2 and θ_1 simply a constraint with respect to event -- "event" semantics do not allow a π node
$\sigma_1\beta\ \chi_1\theta_1$	σ_1 terminates/meets χ_1 and θ_1 -causal action and constraint - event-based control
$\sigma_1\beta\ \chi_1\chi_2$	σ_1 terminates/meets χ_1 and χ_2 which both initiate concurrently - multiple causation
$\sigma_1\beta\ \theta_1\theta_2$	σ_1 terminates/meets θ_1 and θ_2 multiple constraints on σ_1 - a reference event
$\chi_1\beta\ \sigma_1\sigma_2$	χ_1 causes σ_1 and σ_2 which both initiate concurrently - χ_1 <u>causes</u> a cobegin
$\chi_1\beta\ \sigma_1\chi_2$	χ_1 causes σ_1 and χ_2 which both initiate concurrently - χ_1 and χ_2 form a causal chain

Table 3-1: Semantics of Degree Three Beta Nodes.

InNode Out1Out2	Description
$\chi_1\beta\ \sigma_1\theta_1$	χ_1 causes σ_1 and θ_1 constrains the event
$\chi_1\beta\ \chi_2\theta_1$	INCOMPLETE
$\chi_1\beta\ \chi_2\chi_3$	χ_1 causes χ_2 and χ_3 one causal action results in two concurrent causal actions
$\chi_1\beta\ \theta_1\theta_2$	INCOMPLETE
$\theta_1\beta\ \sigma_1\sigma_2$	θ_1 constrains σ_1 and σ_2 which both initiate concurrently - θ_1 constrains a cobegin
$\theta_1\beta\ \sigma_1\chi_1$	θ_1 constrains σ_1 and χ_1 which both initiate concurrently
$\theta_1\beta\ \sigma_1\theta_2$	θ_1 constrains initiation of σ_1 with respect to its predecessor and θ_2 constrains σ_1 with respect to its successor
$\theta_1\beta\ \chi_1\theta_2$	θ_1 constrains initiation of χ_1 with respect to its predecessor and θ_2 constrains χ_1 with respect to its successor
$\theta_1\beta\ \chi_1\chi_2$	θ_1 constrains χ_1 and χ_2, a single constraint on the initiation of two concurrent causal actions
$\theta_1\beta\ \theta_2\theta_3$	A composite constraint -links multiple successors

Table 3-1, concluded

InNode Out1Out2	Description
$\sigma_1\gamma\ \sigma_2\sigma_3$	σ_1 terminates; σ_2 or σ_3 is initiated - a conditional branch with a mutually exclusive selection - (Note: the predicates are not shown.)
$\sigma_1\gamma\ \sigma_2\chi_1$	σ_1 terminates; σ_2 or χ_1 is initiated - a branch point: continue sequence or cause another action - used in model of asynchrony
$\sigma_1\gamma\ \sigma_2\theta_1$	σ_1 terminates; σ_2 is initiated or θ_1 constrains the termination of σ_1
$\sigma_1\gamma\ \chi_1\theta_1$	σ_1 terminates; χ_1 causes another action or θ_1 constrains the termination of σ_1
$\sigma_1\gamma\ \chi_1\chi_2$	σ_1 terminates; χ_1 or χ_2 is initiated - "control" - selection of a coroutine
$\sigma_1\gamma\ \theta_1\theta_2$	σ_1 terminates; θ_1 or θ_2 constrain σ_1 - exclusion of a region in time
$\chi_1\gamma\ \sigma_1\sigma_2$	χ_1 causes σ_1 or σ_2 initiation -select coroutine
$\chi_1\gamma\ \sigma_1\chi_2$	χ_1 causes χ_2 to continue causal chain or causes sequence σ_1

Table 3-2: Semantics of Degree Three Gamma Nodes.

InNode Out1Out2	Description
$\chi_1\gamma\ \sigma_1\theta_1$	χ_1 causes σ_1, or θ_1 constrains the termination of χ_1 - an "inserted process delay" or wait
$\chi_1\gamma\ \chi_2\theta_1$	χ_1 causes χ_2 to continue the causal chain, or θ_1 - constrains the termination of χ_1 - wait/delay
$\chi_1\gamma\ \chi_2\chi_3$	χ_1 causes χ_2 or χ_3 one causal action results in selection of one of two causal actions
$\chi_1\gamma\ \theta_1\theta_2$	INCOMPLETE - No successor (i.e. σ or π arc) ==> No causality
$\theta_1\gamma\ \sigma_1\sigma_2$	θ_1 constrains σ_1 or σ_2 the alternative sequences
$\theta_1\gamma\ \sigma_1\chi_1$	θ_1 constrains σ_1 or χ_1 the alternative sequences
$\theta_1\gamma\ \sigma_1\theta_2$	θ_1 constrains initiation of σ_1 with respect to its predecessor or forms constraint chain with θ_2
$\theta_1\gamma\ \chi_1\theta_2$	θ_1 constrains initiation of χ_1 with respect to its predecessor or forms constraint chain with θ_2
$\theta_1\gamma\ \chi_1\chi_2$	θ_1 constrains χ_1 or χ_2 the causal action branches
$\theta_1\gamma\ \theta_2\theta_3$	Either θ_1 and θ_2 form a constraint chain or θ_1 and θ_3 form a constraint chain

Table 3-2, concluded

In1In2Node Out	Description
$\sigma_1\sigma_2\mu$ σ_3	Upon cotermination of σ_1 and σ_2, σ_3 is initiated -a delay or wait in σ_1 or σ_2 may be required to achieve cotermination
$\sigma_1\chi_1\mu$ σ_2	χ_1 and cotermination of σ_1 enables the initiation of σ_2 - i.e. σ_1 "waits" for χ_1 - or if χ_1 precedes σ_2 initiation is enabled and begins on termination of σ_1
$\sigma_1\theta_1\mu$ σ_2	θ_1 constrains termination of σ_1 and initiation of σ_2; the θ_1 constraint must be satisfied
$\chi_1\theta_1\mu$ σ_1	θ_1 constrains the initiation of σ_1 caused by χ_1; the θ_1 constraint must be satisfied
$\chi_1\chi_2\mu$ σ_1	χ_1 and χ_2 cause initiation of σ_1 - σ_1 waits for both causal actions - or - if χ_1 precedes χ_2 or χ_2 precedes χ_1 - predecessor enables successor
$\theta_1\theta_2\mu$ σ_1	σ_1 initiation is constrained by θ_1 and θ_2 - both constraints must be satisfied
$\sigma_1\sigma_2\mu$ χ_1	σ_1 and σ_2 coterminate; initiating a causal action - cotermination is ensured by inserted delays
$\sigma_1\chi_1\mu$ χ_2	χ_1 and σ_1 coterminate and cause the initiation of the causal action χ_2
$\sigma_1\theta_1\mu$ χ_1	θ_1 constrains the termination of σ_1 and the initiation of χ_1 -constraint θ_1 must be satisfied

Table 3-3: Semantics of Degree Three Mu Nodes.

In1In2Node Out	Description
$\chi_1\theta_1{}^{\mu}\ \chi_2$	INCOMPLETE
$\chi_1\chi_2{}^{\mu}\ \chi_3$	χ_1 and χ_2 coterminate - initiating χ_3 - conjunctive causation - "wait" / delay for both
$\theta_1\theta_2{}^{\mu}\ \chi_1$	θ_1 and θ_2 constrain the initiation of χ_1 - both constraints must be satisfied
$\sigma_1\sigma_2{}^{\mu}\ \theta_1$	Cotermination of σ_1 and σ_2 is constrained by θ_1
$\sigma_1\chi_1{}^{\mu}\ \theta_1$	Cotermination of σ_1 and χ_1 is constrained by θ_1
$\sigma_1\theta_1{}^{\mu}\ \theta_2$	θ_1 constrains inititation of σ_1 with respect to its predecessor and θ_2 constrains σ_1 with respect to its successor
$\chi_1\theta_1{}^{\mu}\ \theta_2$	θ_1 constrains inititation of χ_1 with respect to its predecessor and θ_2 constrains χ_1 with respect to its successor
$\chi_1\chi_2{}^{\mu}\ \theta_1$	INCOMPLETE
$\theta_1\theta_2{}^{\mu}\ \theta_3$	Conjoined constraints; θ_1 and θ_2 constrain θ_3

Table 3-3, concluded

In1In2Node Out	Description
$\sigma_1\sigma_2\rho\ \sigma_3$	Either σ_1 terminates or σ_2 terminates - (exclusive or) therefore sequence is $\sigma_1\sigma_3$ or $\sigma_2\sigma_3$
$\sigma_1\chi_1\rho\ \sigma_2$	Either σ_1 terminates or χ_1 terminates - (exclusive or) therefore sequence is $\sigma_1\sigma_2$ or $\chi_1\sigma_2$
$\sigma_1\theta_1\rho\ \sigma_2$	Either σ_1 terminates or θ_1 constrains the initiation of σ_2 - (exclusive or) therefore $\sigma_1\sigma_2$ occurs or θ_1 constrains initiation of σ_2
$\chi_1\theta_1\rho\ \sigma_1$	Either χ_1 causes initiation of σ_1 or θ_1 constrains the initiation of σ_1 - (exclusive or) therefore the sequence $\chi_1\sigma_1$ occurs or $\theta_1\sigma_1$
$\chi_1\chi_2\rho\ \sigma_1$	Either χ_1 causes initiation of σ_1 or χ_2 causes initiation of σ_1 - (exclusive or)
$\theta_1\theta_2\rho\ \sigma_1$	Either θ_1 constrains initiation of σ_1 or θ_2 constrains the initiation of σ_1 -(exclusive or)
$\sigma_1\sigma_2\rho\ \chi_1$	Either σ_1 terminates or σ_2 terminates, initiating a causal action χ_1 (exclusive or) therefore sequence is $\sigma_1\chi_1$ or $\sigma_2\chi_1$
$\sigma_1\chi_1\rho\ \chi_2$	Either σ_1 terminates initiating χ_2 or χ_1 causes initiation of the causal action χ_2 (exclusive or)

Table 3-4: Semantics of Degree Three Rho Nodes.

In1In2Node Out	Description
$\sigma_1\theta_1\rho\ \chi_1$	Either σ_1 terminates initiating χ_1 or θ_1 constrains the initiation of χ_1 -(exclusive or)
$\chi_1\theta_1\rho\ \chi_2$	Either χ_1 causes initiation of χ_2 or θ_1 constrains the initiation of χ_2 -(exclusive or)
$\chi_1\chi_2\rho\ \chi_3$	Either χ_1 causes initiation of χ_3 or χ_2 causes - initiation of χ_3 -disjunctive causation
$\theta_1\theta_2\rho\ \chi_1$	Either θ_1 constrains the initiation of χ_1 or - θ_2 constrains the initiation of χ_1
$\sigma_1\sigma_2\rho\ \theta_1$	Either σ_1 terminates or σ_2 terminates - θ_1 constrains the termination of σ_1 or σ_2
$\sigma_1\chi_1\rho\ \theta_1$	Either σ_1 terminates or χ_1 terminates - θ_1 constrains the termination of σ_1 or χ_1
$\sigma_1\theta_1\rho\ \theta_2$	Either θ_2 constrains the termination of σ_1 or θ_1 forms a constraint chain with θ_2
$\sigma_1\chi_1\rho\ \theta_1$	Either σ_1 terminates or χ_1 terminates - θ_1 constrains the termination of σ_1 or χ_1
$\chi_1\chi_2\rho\ \theta_1$	INCOMPLETE
$\theta_1\theta_2\rho\ \theta_3$	Disjoint constraints; θ_1 or θ_2 constrain θ_3.

Table 3-4, concluded

3.4. Types of Interspace Bindings

There are two basic types of bindings:[8]

1. **operation** bindings, which relate dataflow elements to structural elements and time ranges, and

2. **realization** bindings, which relate structural elements to physical elements.

These bindings may be further classified based by the type of elements that they relate. There are two subtypes of operation bindings:

1. **value-carrier-range** (vcr) bindings that denote the association of value nodes in the DFss to time ranges in the TSss and also the association of these values to carriers in the Structural subspace (Sss), and

2. **operation-module-range** (omr) bindings that denote the association of operation nodes in the DFss to time ranges in the TSss and also the association of these operations to modules in the Sss.

There are two subtypes of realization bindings:

1. **module-block** (mb) bindings that establish a correspondence between a module in the Sss and a block in the Physical subspace (Pss), and

2. **carrier-net** (cn) bindings that establish a correspondence between a carrier in the Sss and a net in the Pss.

---

[8] These naming conventions follow the naming convention used in other work [Arfarmanesh 85]; however, refinements to the semantics of the DFss and TSss make some details of the bindings inconsistent between this thesis and previous work.

3.5. Control in the DFss

In Section 3.2 the data flow operations of select and distribute [Davis 82] were mentioned but not described. These operations allow us to describe complex sequencing operations in the DFss; essentially, these are the control operations in our data flow model. The **select** operation accepts three inputs and produces one output. One of the inputs is the control input and only accepts a boolean value. If this value is true (T) then the data value corresponding to the T-input appears at the output as shown in Figure 3-16. If this value is false (F) then the alternate input, *i.e.* the one corresponding to the F-input, appears at the output. Thus the value node in the DFss associated with the output of a select operation is not determined until the control input and the selected data input are both known. The **distribute** operation accepts two inputs, a data input and a control input, and produces two outputs, one which corresponds to the input value and an undefined value on the alternate output. Again the control input is Boolean as shown in Figure 3-17.

Generalizations of the selector and distributor are easily devised:

- selection (or distribution) may be based on an integer control value, or
- the inputs and outputs may be replaced by groups of inputs and outputs instead of a single value.

Another possibility is to define degenerate cases of the select operation; namely, the **T-gate** and **F-gate**. These are select operations with one of their inputs being the undefined value, *bottom*. For example a T-gate only *passes a value* when the control input is true; when the control input is false the value at the output is *bottom*.

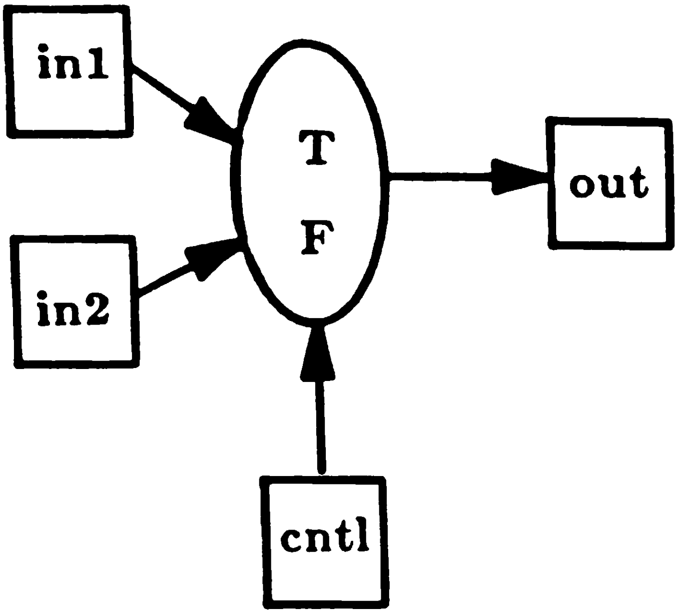


Figure 3-16: A data flow *select* operation.

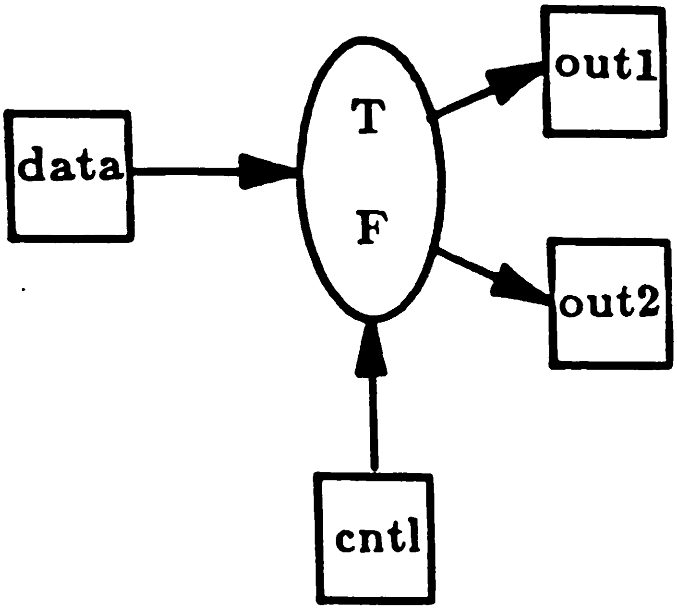


Figure 3-17: A data flow *distribution* operation.

3.5.1. A DDS representation of "while"

Figure 3-18 shows the representation of the Pascal *while* construct [Jensen 85]. In the DDS representation of the *while* construct, the *BooleanExpression* is represented as *Bool* and the *Statement* is replaced by the operation labelled *foo(i)*.

The EBNF (Extended Backus Naur Format) is

WhileStatement = "while" *BooleanExpression* "do" *Statement*

The statement is repeatedly executed until the expression becomes false. If its value is false at the beginning, the statement is not executed at all.

The DDS representation describes this construct using an alpha-omega loop in the TSss together with a select operation and T-gate in the DFss. By "walking the graphs" in the TSss and DFss, we can simulate the behavior of the while construct. On the first iteration $i=1$, and the value c(1) is the same value as b(0). Note b(0) only exists on the interval σ_0 before the main body of the "while" (α-ω loop). Since $i=1$, the nodes are α_1, γ_1, ω_1 and π_1, and if *Bool* is false then the lower branch in the figure, σ_{31} is taken at γ_1 and the final value for b(I) is the value of b(0). On the other hand, if *Bool* is true then the upper branch, σ_{21} is taken, the operation *foo* is executed and c(1) is transformed into the new value b(1). At this time the value b(i-1) is b(1) and $i=2$ so c(2) is the same value as b(1). Let us assume the value of *Bool* is now false, and that the lower branch is taken from γ_2, the value of b(I) is the value of b(1).

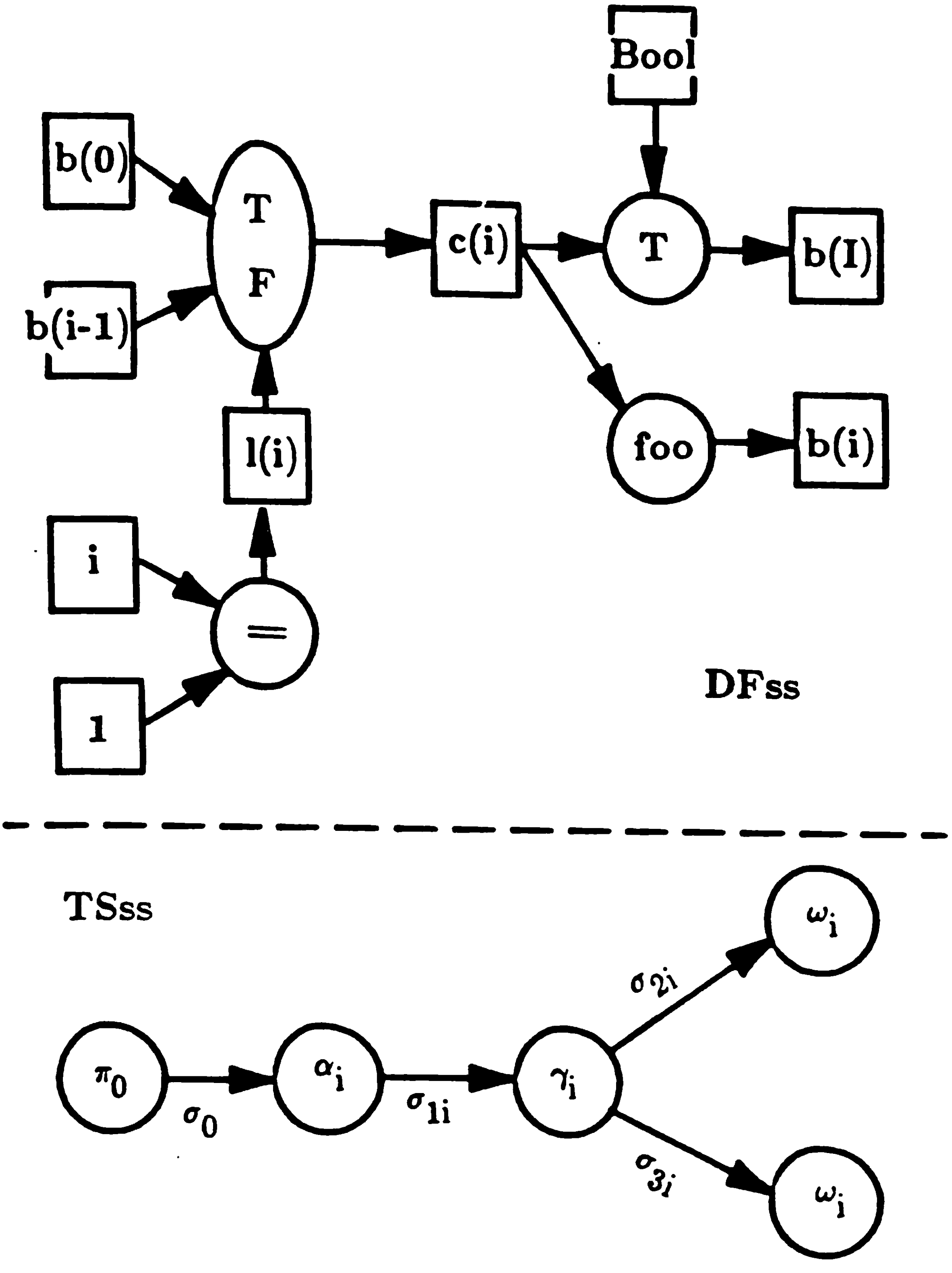


Figure 3-18: A DDS template of the behavior of the Pascal *while* construct.

Bool = F

value	range
b(0)	σ_0
c(1)	σ_{11}

Bool = F

operation	range
Select(1)	σ_0
T-Gate(1)	σ_1, σ_{11}, σ_{31}

Bool = T

value	range
b(0)	σ_0
c(1)	σ_{11}
b(1)	σ_{21}
c(n>1)	σ_{1n}
b(n>1)	σ_{2n}

Bool = T

operation	range
Select(1)	σ_0, σ_{11}
T-gate(1)	σ_{21}
foo(1)	σ_{21}
T-gate(n>1)	σ_{2n}
foo(n>1)	σ_{2n}

Bool = F

value	range
b(n>2)	σ_{1n}, σ_{3n}
c(n>2)	σ_{1n}

Table 3-5: Bindings for the *while* construct shown in Figure 3-18.

3.5.2. A DDS representation of "repeat"

Figure 3-19 and Table 3-6 shows the representation of the Pascal *repeat* construct [Jensen 85]. In the DDS representation of the *repeat* construct, the *BooleanExpression* is replaced by *Bool* and the *StatementSequence* is replaced by the operation labelled *foo(i)*. The EBNF is

RepeatStatement = "repeat" ***StatementSequence***
"until" ***BooleanExpression***.

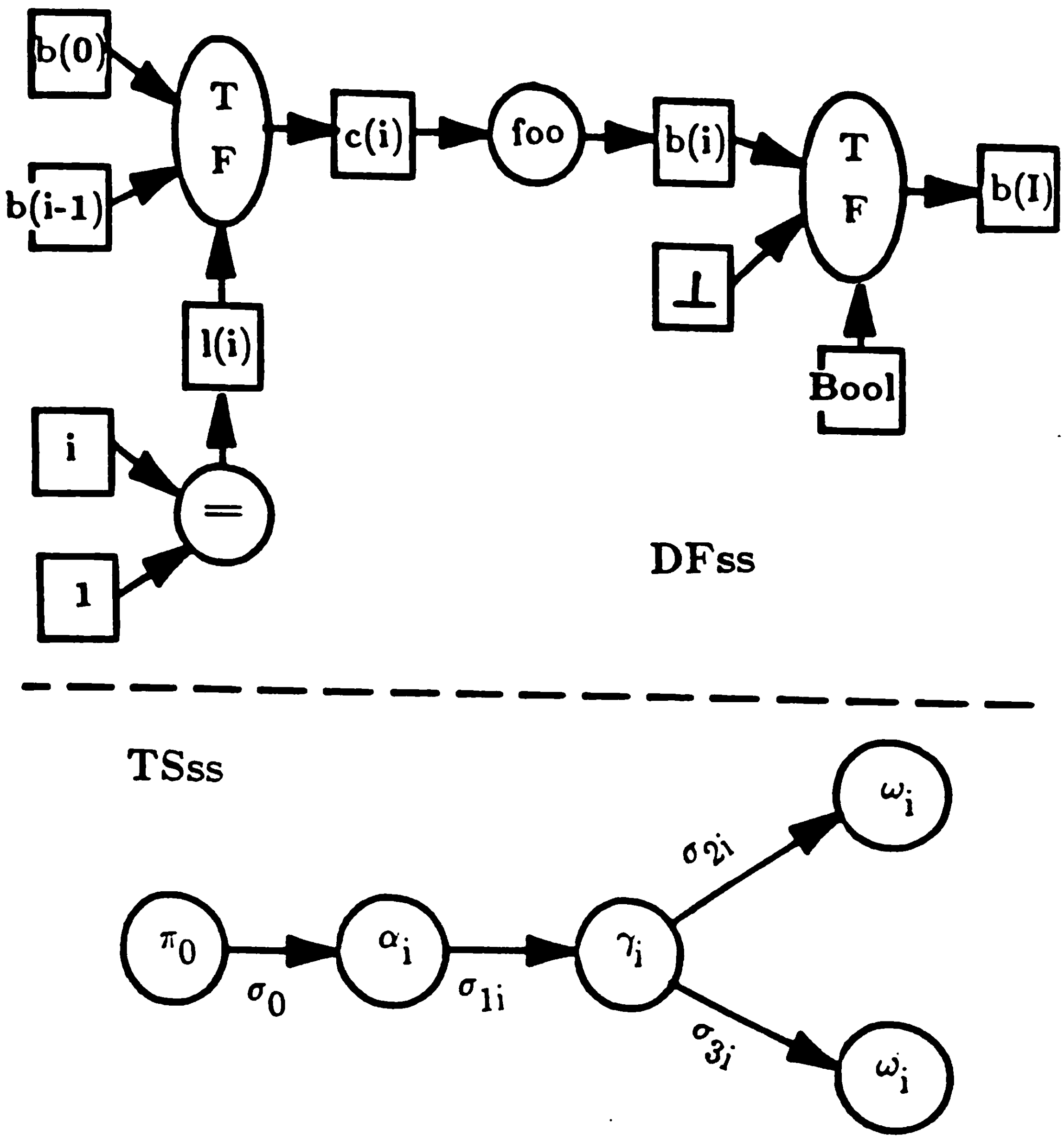


Figure 3-19: A DDS template of the behavior of the Pascal *repeat* construct.

Bool = F

value	range
b(0)	σ_0
c(1)	σ_{11}

Bool = F

operation	range
l-Select(1)	σ_0
T-Gate(1)	$\sigma_1, \sigma_{11}, \sigma_{31}$
Bool-Select(1)	σ_{11}, σ_{31}

Bool = T

value	range
b(0)	σ_0
c(1)	σ_{11}
b(1)	σ_{21}
c(n>1)	σ_{1n}
b(n>1)	σ_{2n}

Bool = T

operation	range
l-Select(1)	σ_0, σ_{11}
Bool-Select(1)	σ_{11}, σ_{21}
foo(1)	σ_{11}, σ_{21}
l-Select(n>1)	σ_{1n}, σ_{2n}
foo(n>1)	σ_{1n}, σ_{2n}

Bool = F

value	range
b(n>1)	σ_{1n}, σ_{3n}
c(n>1)	σ_{1n}

Table 3-6: Bindings for the *repeat* construct shown in Figure 3-19.

The statement sequence is repeatedly executed (and at least once) until the expression becomes true.

The DDS representation describes this construct using an alpha-omega loop in the TSss together with two select operations. By "walking the graphs" as we did for the "while" construct, we can show that this DDS representation has the same behavior as the "repeat" construct. Starting with $i=1$, we see that the value c(1) is the same value as b(0) and the operation *foo(1)* produces a new value, b(1). If *Bool* is true then the loop is exited at γ_1 on the first iteration and b(I) has the same value as b(1). If *Bool* is false, then b(1) is undefined on the upper branch, and the loop is executed again with c(2) assuming the value of b(1) and the operation *foo* produces a new value b(2). This would continue until the predicate is false and the lower branch of the γ node would be taken.

3.6. Hierarchy in the DDS

Each subspace in the DDS supports the notion of hierarchy, *i.e.* every object in any of the spaces may be either primitive or structured with one important exception, points in the TSss.

Points or events in the TSss are dimensionless and therefore cannot be decomposed. Ranges are not subject to this restriction because they represent the passage of time which clearly can be decomposed into sequences of ranges.

3.6.1. Rules of composition for the DFss, Sss and Pss Hierarchies

Some important rules of the various hierarchies are the following:

1. Recursion is not permitted. Some bound recursion can be described in terms of multiple hardware units or iteration, but hardware in general does not support recursion.

2. Data flow operations may be composed of data link arcs, data flow values, and other data flow operations.

3. Data flow values and their associated data link arcs may be composed of other data link arcs and data flow values. For example a complex floating point number can be represented as a *real* and an *imaginary* part, *i.e.* two data flow values and two sets of data flow arcs. The real and imaginary parts can be further decomposed into an *exponent* and a *mantissa*. Structured data flow values do not contain data flow operations or data link arcs.

4. Modules in the structural subspace may be composed of carriers and other modules.

5. Carriers in the structural subspace may be composed of other carriers.

6. Blocks in the physical subspace may be composed of nets and other blocks.

7. Nets in the physical subspace may be composed of other nets.

3.6.2. Rules of composition for the TSss

The hierarchical description of the TSss is much more complicated than the other subspaces because of the many different node types, arc types and their associated predicates. This situation is compounded by the fact that points (*i.e.* events) are not composite objects and therefore cannot be decomposed.

3.6.2.1. Pseudo composite events

Not allowing events to be composite objects creates problems in the hierarchical interpretation of the TSss as well as difficulty in describing some *chains* of events at a single level in the TSss. For example, when the asynchronous predicate branches to a destination point this must be a rho node. This rho node has only one outgoing arc and cannot model the start of two concurrent ranges.[9] To model this sequence of events, the events are connected by constraint arcs that have a length equal to zero as shown in Figure 3-20. The interpretation of a sequence of events connected only by constraint arcs that have a length equal to zero is that one cannot resolve the amount of time between any two events or among the events in a

---

[9] A new node type might be defined but as these *chains* get more complex the number of node types would grow and make it difficult to develop tools to check the validity of the structures created.

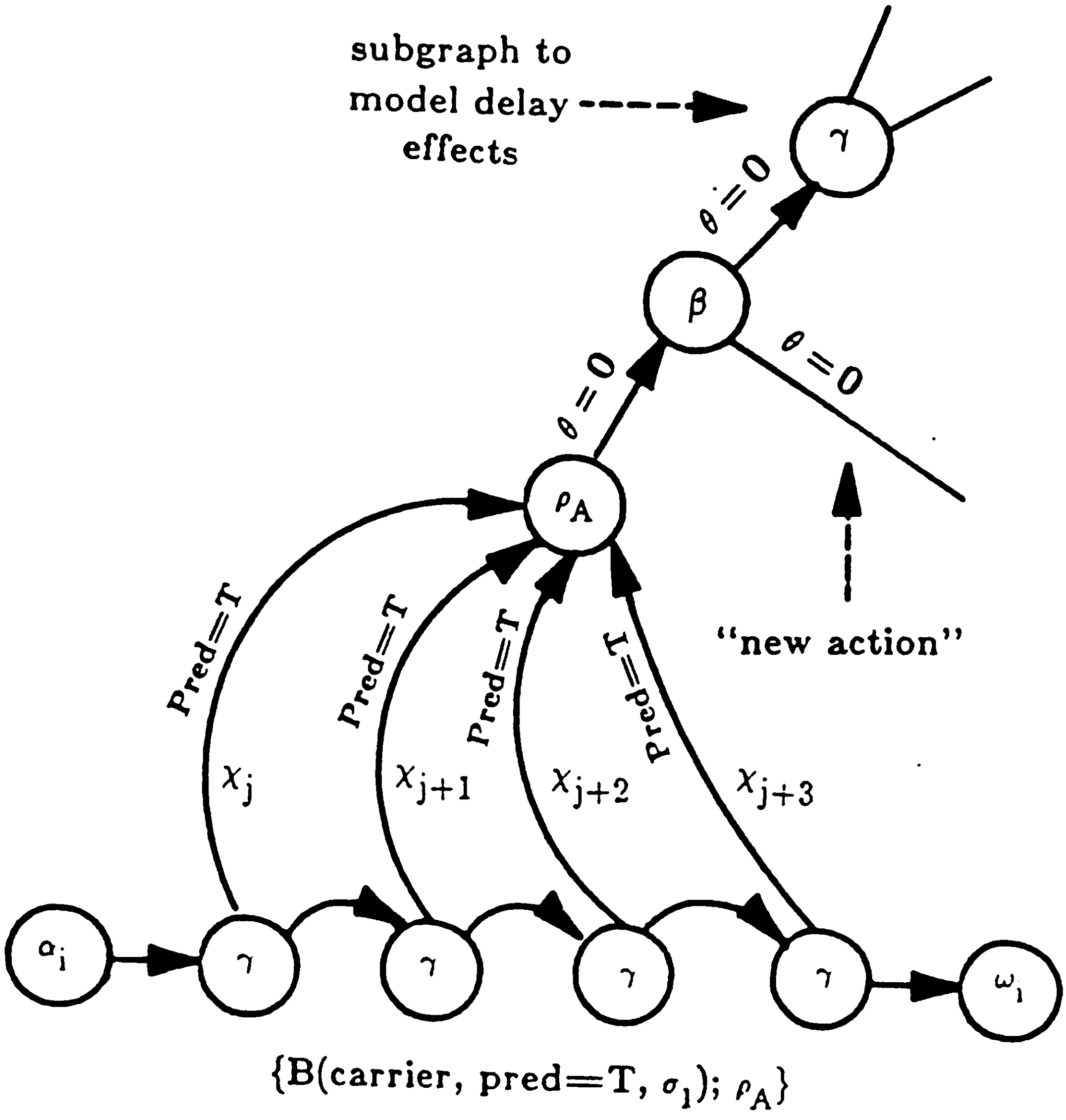


Figure 3-20: An example of using constraint arcs to extend the destination node of an asynchronous branch.

longer chain. This allows us to construct arbitrarily complex nodes by using only nodes of degree less than or equal to three. For example, a beta node that requires five output arcs can be modeled as shown in Figure 3-21. The interpretation of this graph in the TSss is that the the time range associated with the incoming arc to the first beta node *meets* the outgoing arcs labelled σ_1 through σ_5. Hence, all types of complex events can be modeled as a sequence of nodes of degree three or less joined by constraint arcs that have a length equal to zero. These arcs, as used here, serve the same purpose as the *dummy arcs* introduced by Knapp [Knapp 83]. However, there are no other similarities and the semantics of these constraint arcs are quite different.

3.6.2.2. Composite ranges and TSss-links

If a composite range is decomposed into two sequential ranges, the end points of the composite range have to be related to the component ranges. Unlike the other subspaces. the timing subspace is not self descriptive when hierarchy is introduced. A special **TSss-link** must be introduced that defines this intraspatial-hierarchical relationship. The TSss-link can only link a single event at one level of the TSss hierarchy with a single event at the next level above or below in the TSss hierarchy. The event on the higher level of the hierarchy must be either the same type of event as the event at the lower level or a pi event. Although this may appear to be an overly restrictive constraint, it is necessary to avoid creating complex events. Two events that are linked by a TSss-link are identical in time. Any range may be divided into subranges; however, no attempt is made in the current model to link ranges and subranges at different levels of the hierarchy.

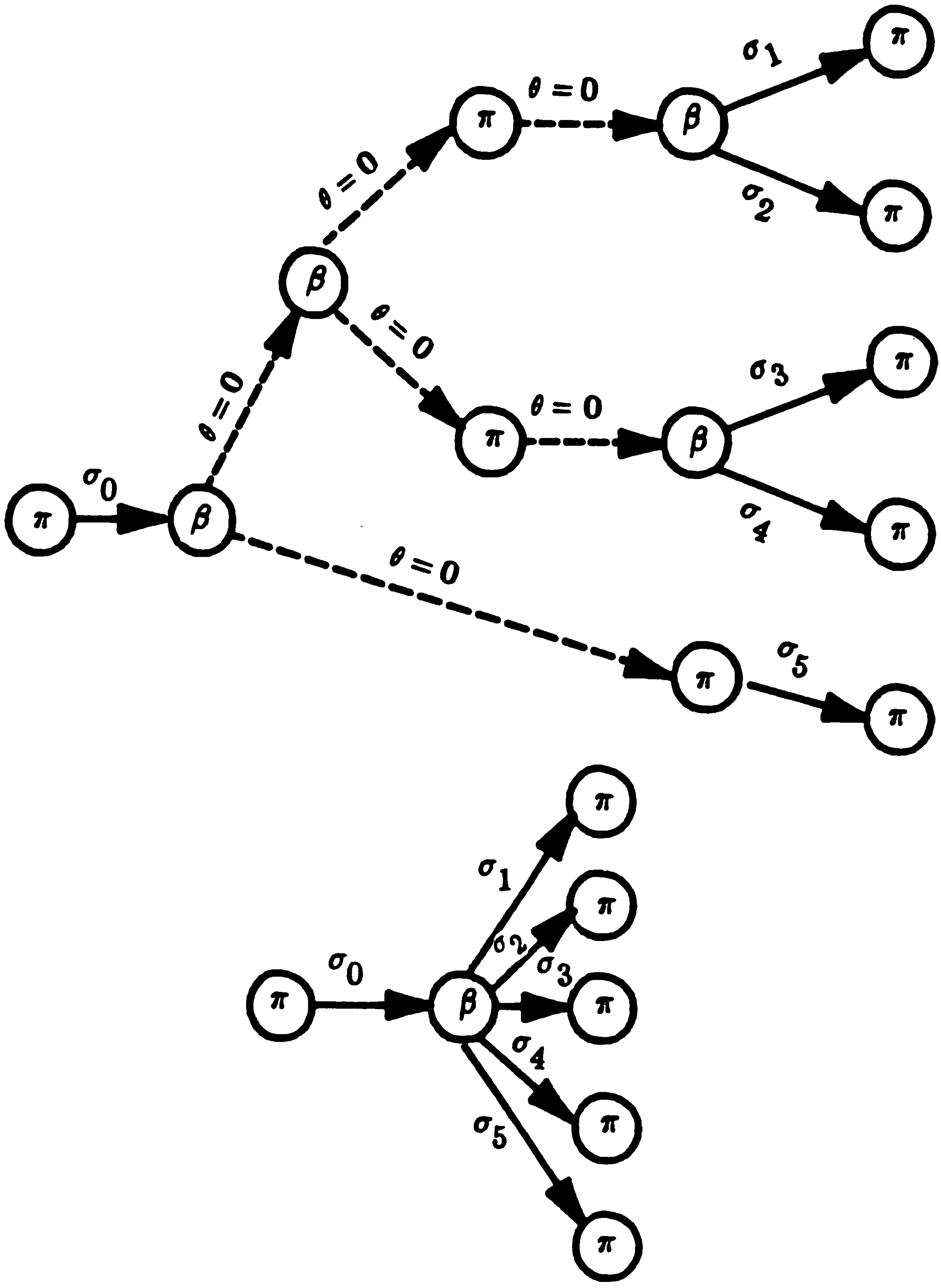


Figure 3-21: Model of a degree five beta node constructed using constraint arcs

3.6.3. Inheritance of predicates in the TSss

Since a synchronous predicate must be true prior to reaching the gamma node at which the predicate determines the action, there is no need to define inheritance for a synchronous predicate. In fact, a synchronous predicate could be reinstated with the opposite value on the chronological successor to the arc following the choice. This is not the case for an asynchronous predicate, which is specifically defined for a given range. For the asynchronous case, all subranges that are part of the range defined in the asynchronous predicate inherit the predicate. Therefore, if the asynchronous predicate is to be disabled in some subrange, an enabling condition must be included in the asynchronous predicate and must be false over the desired subrange. This approach allows multiple predicates to be turned on and off selectively for any set of subranges.

These additions to DDS reflect semantics of ISPS, SLIDE and system-level specifications studied as part of this research. The use of these semantics is described in Chapter 4.

Chapter 4

System Behavior and Its Representation

4.1. Introduction

To understand the specification of digital systems in restricted English text requires

1. a corpus (a collection of writings, in this case examples) for the domain of these specifications,
2. a representation for the knowledge expressed, in the corpus,
3. a formal representation for the behavior of a digital system, and
4. a parsing technique to map the natural language into the formal "behavioral" representation.

This chapter will first discuss the corpus, the knowledge about system specifications contained in the corpus and the parsing technique. Next we will discuss the representations used in this research by analyzing example sentences taken from the corpus for the domain of digital specifications. The knowledge in the corpus is represented in two forms. The first representation is called **abstract behavior**; it is a generalization of system behavior based on groups of sentences taken from natural language specifications. The second representation is the **concept** from the **pattern-concept pair** used by PHRAN. In this chapter, only PHRAN's concepts will be discussed in relationship to the other representations. The formal representation of the behavior is templates in the DDS. Each type of abstract behavior and in some cases PHRAN concepts are formally described by a DDS template. A DDS template is composed of a data flow subgraph and a timing and sequencing subgraph and their interspatial bindings. A specification is formed by combining the various subgraphs where appropriate to form a single graphical representation in the DSS. Prior to discussing the detailed examples, the relationship of the multiple representations will be presented. Next, each type of abstract behavior or PHRAN concept and its related DDS template will be described along with example sentences taken from actual specifications.

4.2. The corpus

The corpus for this natural language interface was developed by acquiring actual specifications, having students write specifications and constructing additional examples. Examples of sentences taken from this corpus are described in this chapter and Chapter 1. The corpus was also used to develop the 2000+ word lexicon used in this research. (The list of vocabulary words is contained in Appendix A.)

4.3. The corpus' knowledge and the parsing technique

The representation of the knowledge expressed in this corpus was constrained by the choice of a pre-existing parsing technique, which was implemented by Arens in PHRAN, a PHRasal ANalysis program [Arens 86].

PHRAN is a knowledge-based approach to natural language processing. The knowledge is stored in the form of pattern-concept pairs. A pattern is a phrasal construct which can be a word, a literal string (*e.g.* **Digital Equipment Corporation**), a general phrase such as

<abstract component><sends><data>to<abstract component>

or may be based on parts of speech, for example,

<noun-phrase><verb>.

Associated with each phrasal pattern is a concept. The pattern-concept pair encodes the lexical, syntactic and semantic knowledge of the language. For example, associated with the pattern:

<abstract component><sends><data>to<abstract component>

is the UVT concept (Section 4.2) that denotes a transfer of data from one component to another component.

The concepts in PHRAN used in this research are expressed in SRL (Specification Representation Language), a representation based on Conceptual Dependencies (CDs) as developed by Schank [Schank 75]. CDs are a declarative representation of meaning which are based on general concepts of human action, human interaction and other generalizations about physical objects. SRL was created to capture the information associated with the specification of digital systems. SRL does not conflict in any known way with Schank's original CDs or any extensions to them used in PHRAN; hence they could be used in conjunction with any system based on the original CD concept.

4.4. The parsing technique

PHRAN reads the sentence from left to right one word at a time. As each word is examined, existing patterns and concepts are checked for a match and retained, modified or discarded. The match may be based on lexical criteria, semantic criteria and/or syntactic criteria. PHRAN also provides some degree of look-ahead in the sentence to the next word and the ability to look back at previously matched terms with some limited ability to modify those previously matched terms.

4.5. Multiple Representations: Abstract Behavior, DDS Templates, and Concepts

The mapping of natural language sentences into the DDS is accomplished by using an intermediate representation, namely, a **concept** from a **pattern-concept pair** used by PHRAN. This intermediate representation captures the *meaning* of a single natural-language sentence. However, specifications of digital systems are composed of groups of sentences, paragraphs, sections, etc. Therefore, a more general representation than **PHRAN's concepts** is required to capture the semantic content of these larger syntactic units. **Abstract behavior** is a generalization derived from groups of sentences taken from natural language specifications--it is the basis for the **DDS template,** as shown in Figure 4-1. In some cases, a single natural language sentence simply describes a DDS template directly; for these cases there is only a PHRAN concept; no higher-level abstract behavior exists.

4.6. The Unidirectional Value Transfer

The abstract behavior of a UVT (Unidirectional Value Transfer) consists of the *transfer* of a value from one operation to another operation in the data flow subspace of the DDS and the associated timing and control information. An example of a sentence that belongs to this category of behavior is:

The cpu sends the data to the memory.

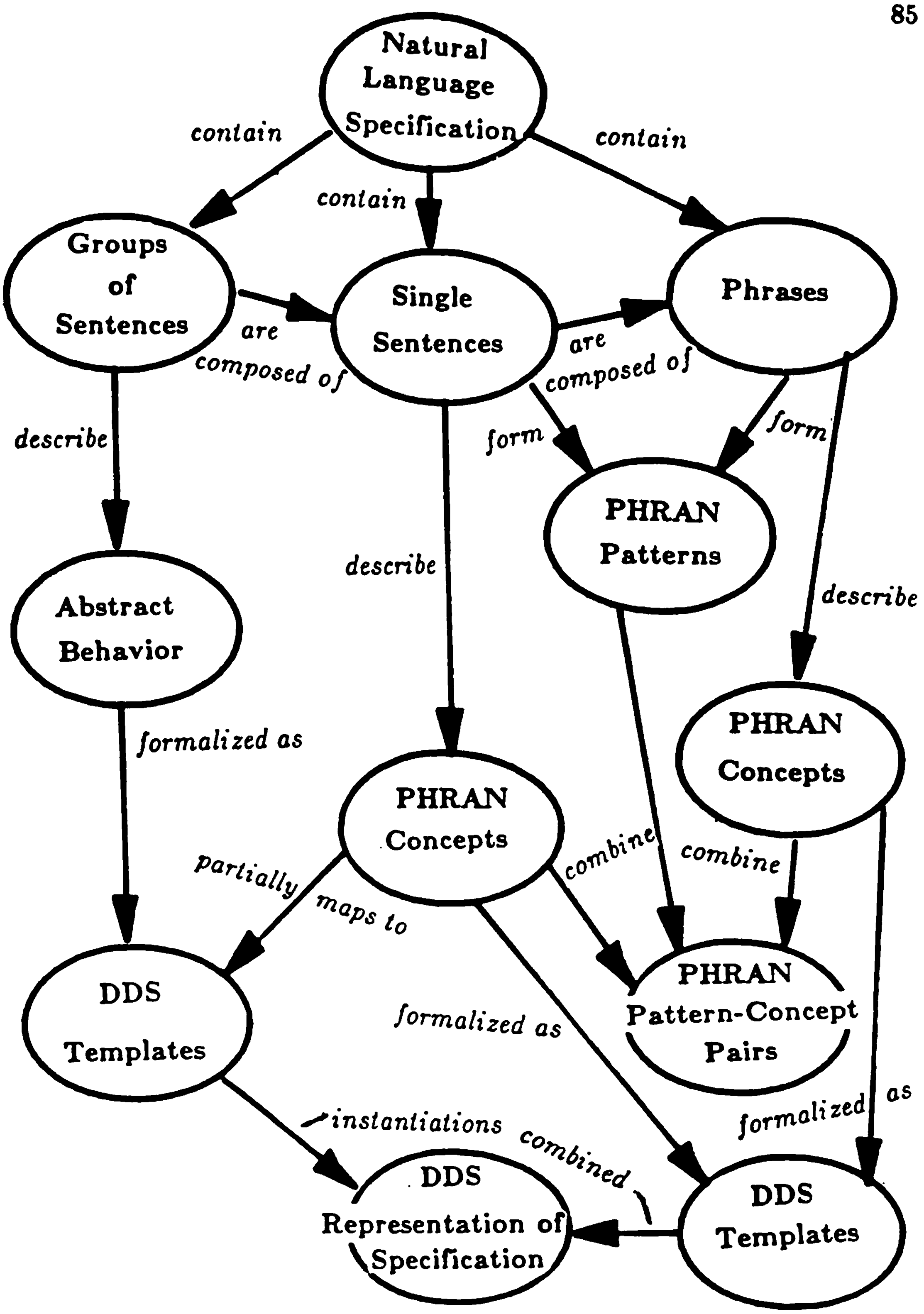


Figure 4-1: The relationship of natural language sentences, abstract behavior, pattern-concept pairs, and DDS templates.

4.6.1. The DDS template for the UVT

The DDS template for the UVT consists of two subgraphs and their associated bindings, as shown in Figure 4-2 and Table 4-1.

This representation of the UVT in the data flow subspace and the timing and sequencing subspace describes the **behavior** associated with this concept.

4.6.1.1. Inclusion of data flow and control flow

The DDS template for the UVT shown in Figure 4-2 includes both data flow information and control flow information in the data flow subspace. The rationale for including three data flow operation nodes in the DDS template for the UVT is that the source or the sink might be different from the operation controlling the transfer. Furthermore, a model that simply has a source and a sink is not capable of modelling a three party transfer where the controlling agent is different from the source or the sink. Additional reasons for including this control flow information include

1. to capture implementation details, *e.g.*, interface specifications, which must be included in the specification, and

2. to provide the *hooks* for attaching the remaining control flow information when the specification is implemented.

In Figure 4-2 the control operation (**cntl**), the source control value (**src cntl**) and the sink control value (**snk cntl**) are all optional and may not impact the completeness of a specification. The other abbreviations used in Figure 4-2 are

1. **src** for the source of the data,

2. **info** for the data flow value transferred, and

3. **snk** for the sink for the data.

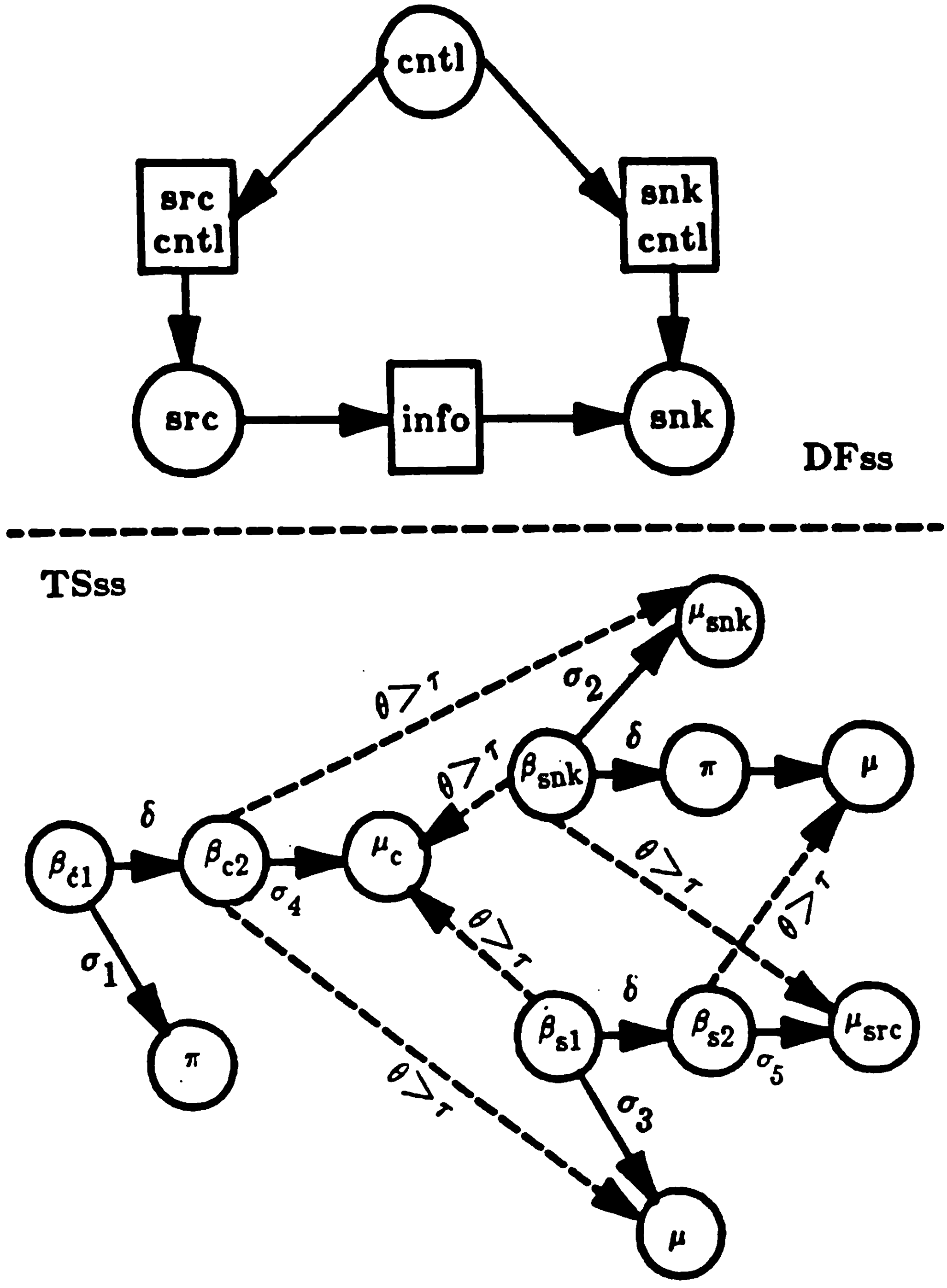


Figure 4-2: The DDS template for a UVT.

value	range	operation	range
src cntl	σ_4	cntl	σ_1
snk cntl	σ_4	src	σ_3
info	σ_5	snk	σ_2

Table 4-1: Bindings for the UVT concept shown in Figure 4-2.

By including the control operation and both a source control value and a sink control value in the DDS template all of the asynchronous control configurations are described by a single DDS template. The four possible one-way asynchronous control configurations and their interpretations are

1. uncontrolled or uncooperative--no control operation node and no control values,
2. one-way source initiated control--the sink control value (DATA READY) and the control operation node are used. (This operation is called source initiated control because the source is the controlling agent and it sends the control value to the destination when it knows the data has been sent and is ready.) [Hayes 78],
3. one-way destination initiated control--the source control value (DATA REQUEST) and the control node are used. (This operation is called destination initiated control because the destination or sink request the data from the source by sending the control value to the source when it is ready to receive the data.) [Hayes 78], and
4. third party control--the source and sink are both sent control information from a third party; *e.g.*, a virtual address might be sent to the source and sink nodes on a token ring by the ring arbiter.

In addition to the control information included in the data flow subspace, various timing constraints can be used independently or added to the four basic control configurations to produce various synchronous or semi-synchronous control schemes.

4.6.2. The UVT Concept--a higher level of abstraction

The **UVT** *concept* used in PHRAN is derived from the abstract behavior of a UVT as described in Section 4.2 and example sentences taken from natural language specifications. The UVT concept focuses on the principal element in the value transfer, namely, the data flow value. Since actual sentences often mix information from the data flow, structural and physical subspaces, operations, logical operators and physical implementations of operators often cannot be distinguished. Thus, the UVT concept replaces the data flow operations found in the abstract behavior by **abstract components**. The actual DDS subspace that an abstract component belongs to could be determined (perhaps during postprocessing) by using additional semantic information found in PHRAN's knowledge base, a previous **declaration** (ref. Section 4.10), additional user input or some combination of this information. Furthermore, this determination cannot usually be extracted directly by PHRAN but would require postprocessing. The UVT concept may then be viewed as a representation in a more abstract space which maps into the DDS. A partial representation and a possible mapping to the DDS template are shown in Figure 4-3. The hexagons in this figure represent the **abstract components** and their potential mappings to DDS subspaces are illustrated by the three dashed arrows and the curved figure used to group them together. The timing is not indicated in the UVT concept, but correct behavior requires the timing and sequencing subgraph shown in Figure 4-2 for each UVT concept. In effect, each specific sentence about value transfer is being isolated by the UVT concept from the details associated with timing and sequencing.

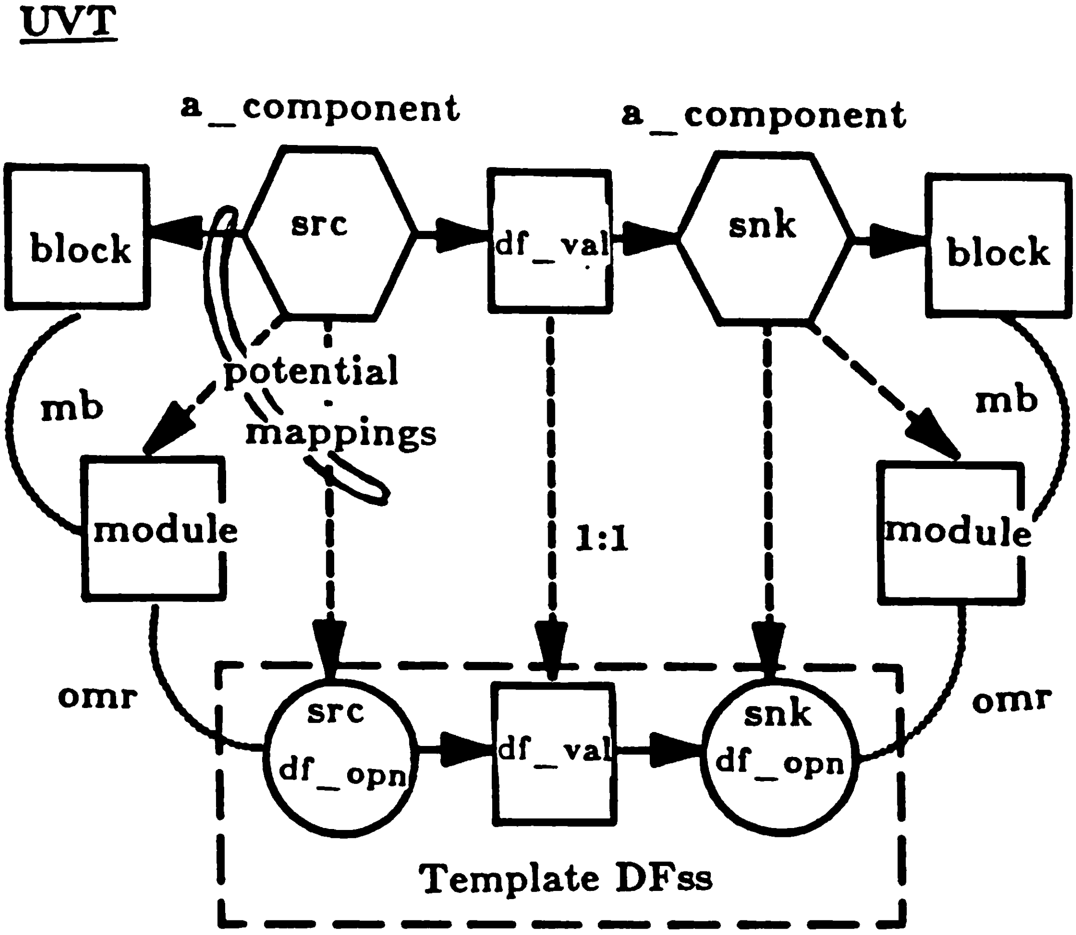


Figure 4-3: The UVT concept and mappings into the DDS template.

4.6.3. Example sentences for the UVT concept

The first example sentence for the UVT concept taken from an actual specification [USN 73] is

The transmitting equipment shall send the word to receiving equipment.

The main verb in this sentence, **send**, expresses the semantic concept of two components, the **transmitting equipment** and the **receiving**

equipment, interchanging information, the **word**. Some other verbs that express a similar semantic concept are **transmit**, **transfer**, **receive**, **return**, **sample**, **signal**, **read**, **write** and **resend**. Each of these verbs denotes an action similar to the action denoted by **send**. When used in a sentence, each of these verbs relates the **control**, a **sink** for the information, and the **information** transferred. Note, this concept has been formulated to reflect only the explicit semantic content of this sentence; hence, we have not included any implied semantic information. An example of implied semantic information is the **source** of the information in the UVT. In most sentences, like the example using **send**, the **source** is not usually expressed and might be erroneously assumed to be the control; however, another example using the verb **transfer** will demonstrate the necessity to have the source included in the concept for completeness. A hypothetical sentence (*i.e.* not taken directly from an actual specification) using **transfer** is:

The cpu transfers the block of data from main memory to the peripheral device.

In this example the cpu is the controlling agent and the source is explicitly stated as the main memory. The complete UVT concept is expressed in the SRL as a frame-like data structure [Winston 84]:

```
(uni_dir_vtrans (source (a_component ?from))
                (sink (a_component ?to))
                (info (df_val ?df_val)
                (control (a_component ?actor)))
```

The *uni_dir_vtrans* frame corresponds to a UVT, the slots in the frame correspond to the objects in the DFss, namely, the *source*, *sink*, *info* and the *control*. Each slot has a facet associated with it that consists of two elements a facet name and a facet value. The facet name *a_component* corresponds to an abstract component and the facet name *df_val* corresponds to a data flow value. The place holders for the facet values are prefixed with a question mark. These place holders are replaced by their specific values in the sentence when PHRAN analyzes the sentence (see the sections discussing PHRAN). All the facet values in this frame are assigned a default value of ***unspecified*** so that if a facet value does not appear explicitly in the sentence SPAN will be able to detect the incomplete information.

As discussed earlier, additional information is required to positively identify the abstract component and the subspace referenced in this sentence--for example, the transmitting equipment could be a module in the structural subspace or a block in the physical subspace or possibly even a data flow operation. Furthermore, if the **transmitting equipment** refers to a library component, then representations in all four subspaces and their associated bindings would be available to complete the specification graph describing the unidirectional value transfer.

The resulting concept produced by PHRAN for the example sentence with the verb **send** is shown:

```
(uni_dir_vtrans (source (a__component *unspecified*))
                (sink (a_component receiving-equipment1))
                (info (df_val word)
                (control (a_component transmitting-equipment1)))
```

This uni_dir_vtrans is represented as a fragment of a graph or subgraph in the data flow subspace and also as fragments of graphs in the structural subspace or the physical subspace depending on additional information that might be available from previous sentences concerning the abstract components. These DDS fragments are shown in Figure 4-4. (See Figure 4-3 for possible mappings of the abstract component to the DDS subspaces.) Since, some words have a restricted semantic meaning when used in a

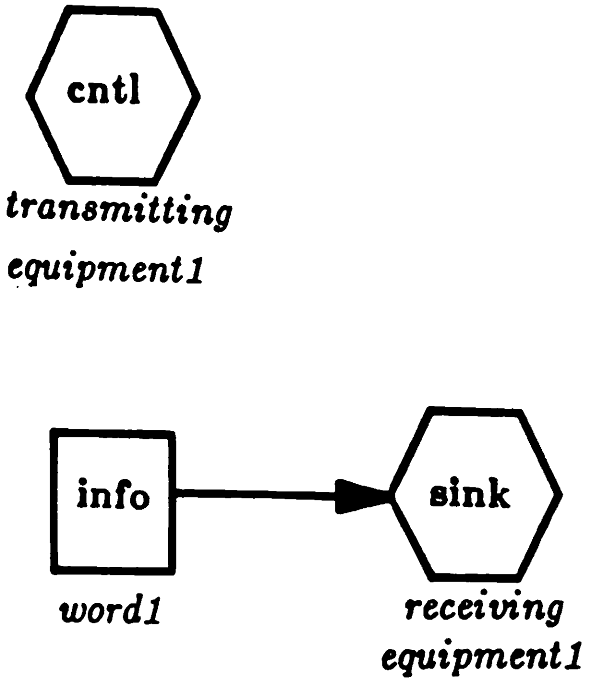


Figure 4-4: The fragments of graphs in the DDS found in the analysis of the example UVT sentence.

specification, it is possible to have a sentence result in a UVT concept that maps directly into the DDS template. This type of sentence describes the data flow **behavior** associated with the UVT. A hypothetical sentence that describes behavior is shown in Figure 4-5, with its corresponding subgraph in the data flow subspace of the DDS. It is difficult to include all the information concerning the abstract behavior of a UVT in a single sentence; hence, the UVT concept only focuses on the data flow information.

Hypothetical sentence:

The arbitration process transfers the priority information from the requesting process to the resolution process.

Data flow subgraph:

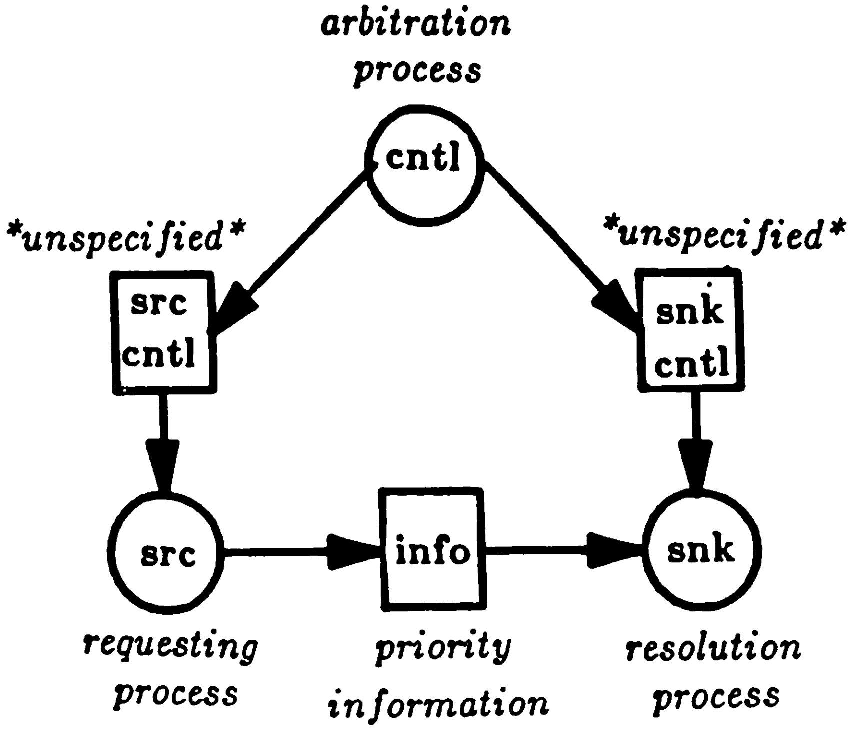


Figure 4-5: A complete dataflow subgraph for a UVT concept.

4.7. The Bidirectional Value Transfer

The abstract behavior of a BVT (Bidirectional Value Transfer) is the *interchange* of values between two operations in the data flow subspace of the DDS. The single assignment nature of the data flow graphs in the DDS requires a pair of values and copies of the operations.

4.7.1. The DDS template for the BVT

The DDS template for the BVT is shown in Figure 4-6. It consists of two UVT representations that are interrelated.[10] This interrelationship arises from the semantics associated with a reciprocal exchange of information. The interrelationship of the sources and sinks in this figure can be better understood by referring to Figure 4-7. In the data flow subgraph bound to com_cycle1, the data flow operation representing the source of info1(i) is src1(i), *i.e.* the abstract component src/snk1 acting as the source node. Also, the abstract component src/snk2 is acting as the destination or sink node, snk2(i). In the data flow subgraph bound to com_cycle2 snk1(j) is src/snk1 acting as the destination or sink node and src2(j) is the abstract component src/snk2 acting as the source node. (Note, the notion of full duplex communication is supported by this DDS template since com_cycle1 and com_cycle2 may be concurrent.)

4.7.2. The BVT concept--a higher level of abstraction

Like the UVT concept, this more abstract concept only requires a value to be referenced in the sentence being analyzed. In sentences associated with this concept the control is almost never mentioned, but the dual role of source and sink for each abstract component is clearly intended. The relationship between this concept and the DDS representation of a BVT is shown in Figure 4-7.

---

[10] The only identification of this reciprocal relationship of sources and sinks is through the names of the operations in the DFss. When these operations are later bound to the same module or block the relationship will be explicitly captured in the DDS.

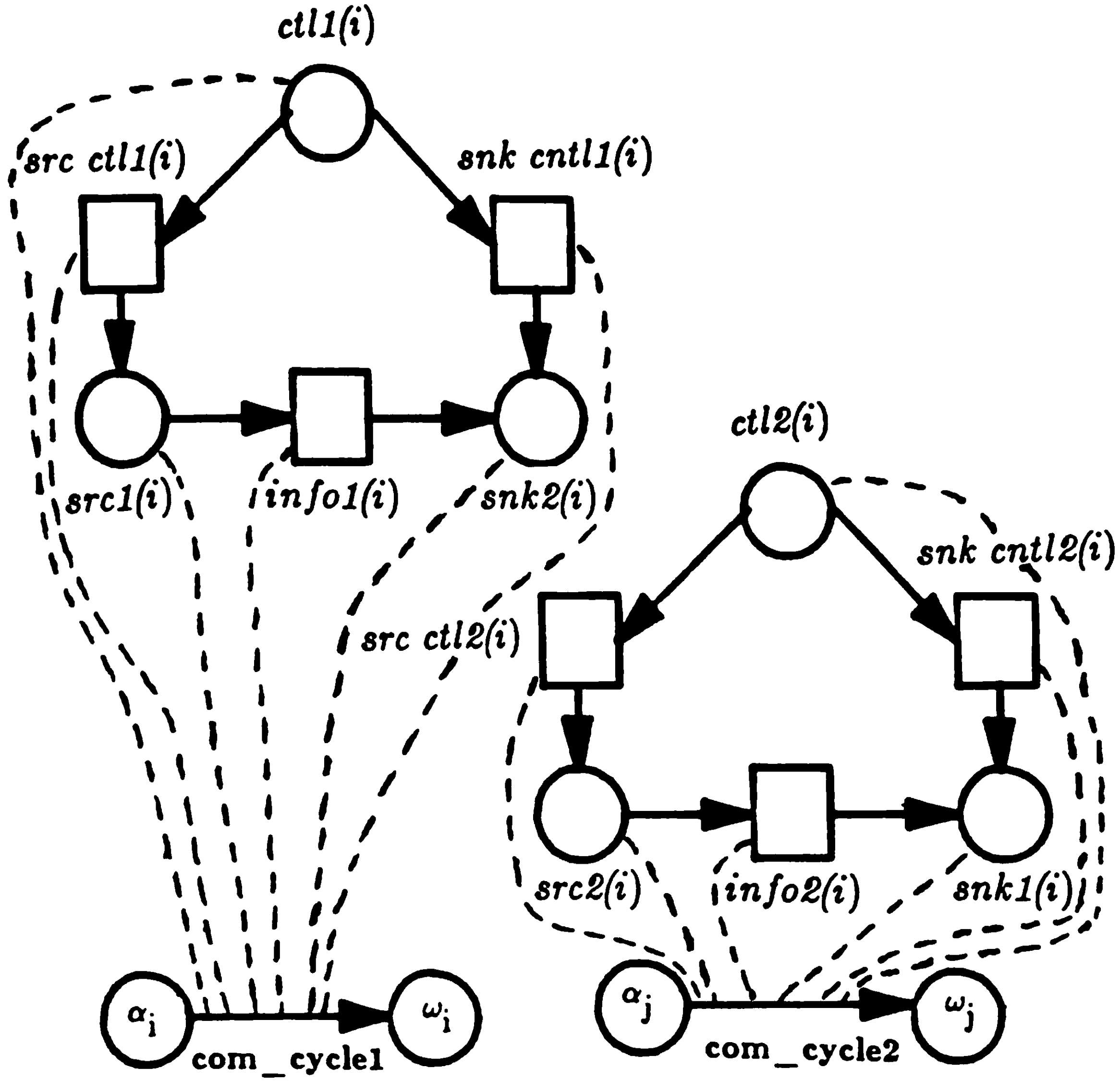


Figure 4-6: The DDS template for a BVT.

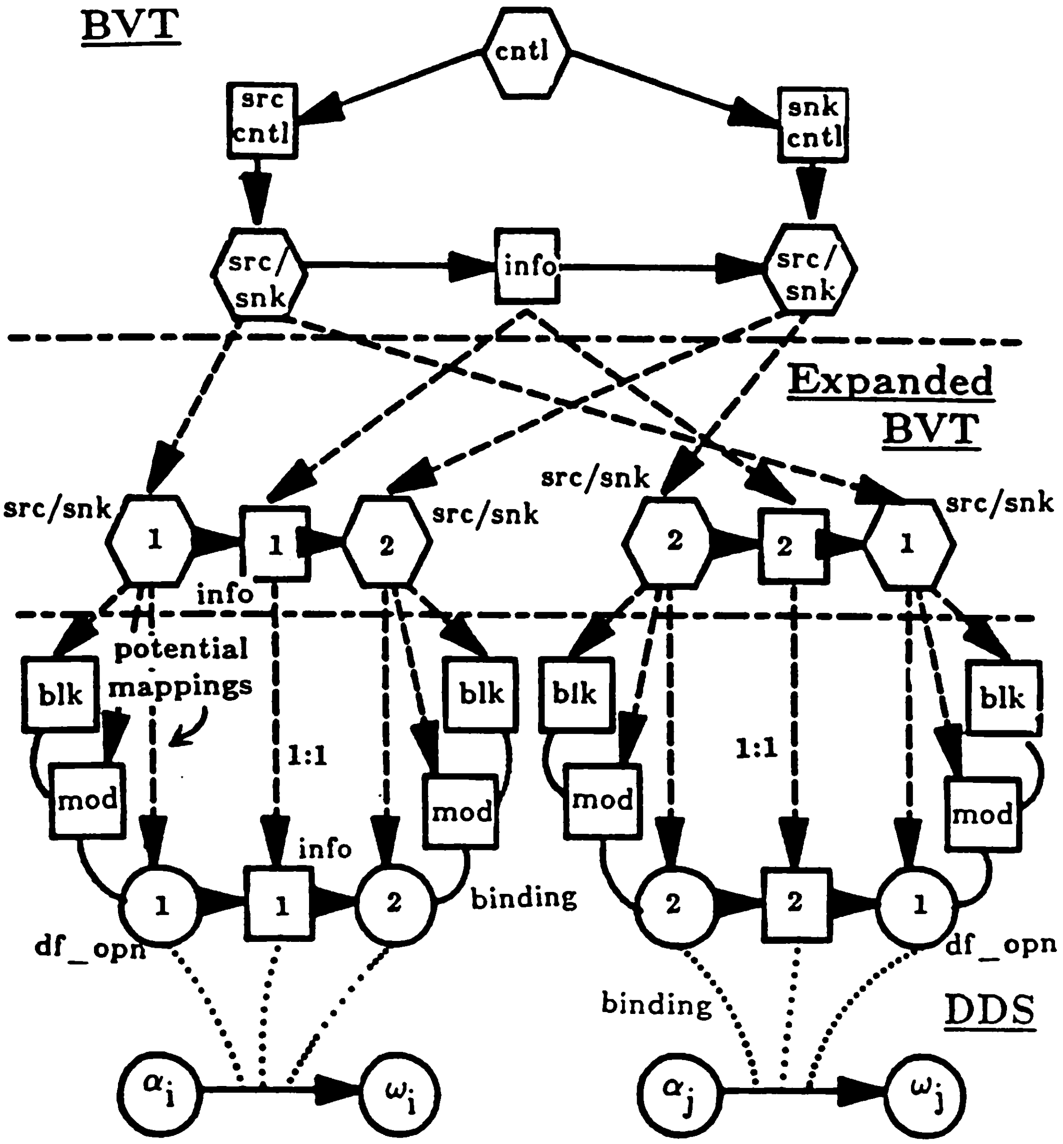


Figure 4-7: The BVT concept and mappings into the DDS template. (Mapping of the control is not shown to simplify the diagram.)

4.7.3. Example sentences for the BVT concept

In many cases, there is a reciprocal exchange of information between various components in a digital system. The following sentence illustrates this semantic concept:

Each requestor communicates with the arbiter via two lines, a request line and a grant line.

The verb **communicate** and the adverbial phrase **with the arbiter** indicates that each requestor acts as a source for information sent to the arbiter and the arbiter acts as a source for each requestor. The phrase **via two lines** and the appositive **a request line and a grant line** are also processed by PHRAN-SPAN but are not essential to this concept and therefore will not be included in this discussion. We focus on the part of the sentence

Each requestor communicates with the arbiter

This concept is fundamentally different from the unidirectional value transfer because of the dual role of the components in the concept. Without additional information the ordering of the value transfers that are described by this concept cannot be determined. Other verbs that express a similar semantic concept are **interchange** and **exchange.** The concept as expressed in this example is very similar to the UVT concept with the **source** and **sink** being replaced by **src/snk**. In fact, the BVT concept could be represented as a pair of UVT concepts with the values in the source and sink slots exchanged and the other information for the value and control repeated. However, this requires information to be derived from the interrelationship between the pair of UVT concepts to capture the notion of communication. Also, the communication may be a sequence of alternating unidirectional value transfers rather than an isolated pair--the bidirectional concept does not make any assumption about the number of value exchanges. The resulting concept is represented as:

```
(bi_dir_vtrans (src/snk1 (a_component ?actor))
               (src/snk2 (a_component ?with))
               (info (df_val ?df_val))
               (control (a_component ?a_component)))
```

After processing the example sentence, PHRAN replaces the slot-fillers like ?actor and ?with, and the resulting concept is

```
(bi_dir_vtrans (src/snk (a_component requestor1))
               (src/snk (a_component arbiter1))
               (info (df_val *unspecified*))
               (control (a_component *unspecified*))
```

The data flow subgraph for a bidirectional transfer with no additional information would be two unidirectional value transfers that have no data precedence relations between them. As in the analysis of the UVT concept no assumptions can be made about the semantic categories for the requestor and arbiter, as shown in Figure 4-8.

In addition to expressing a bididirectional value transfer, **communicate** is unique in that it can also express a unidirectional value transfer by using **to** in the place of **with**. An example of this use of communicate is

The main process communicates the priority values to all the subprocesses.

Other verbs which express unidirectional value transfers may also be used with the phrase **back and forth** to express bidirectional value transfers (*e.g.*, the cpu passes data back and forth to the memory).

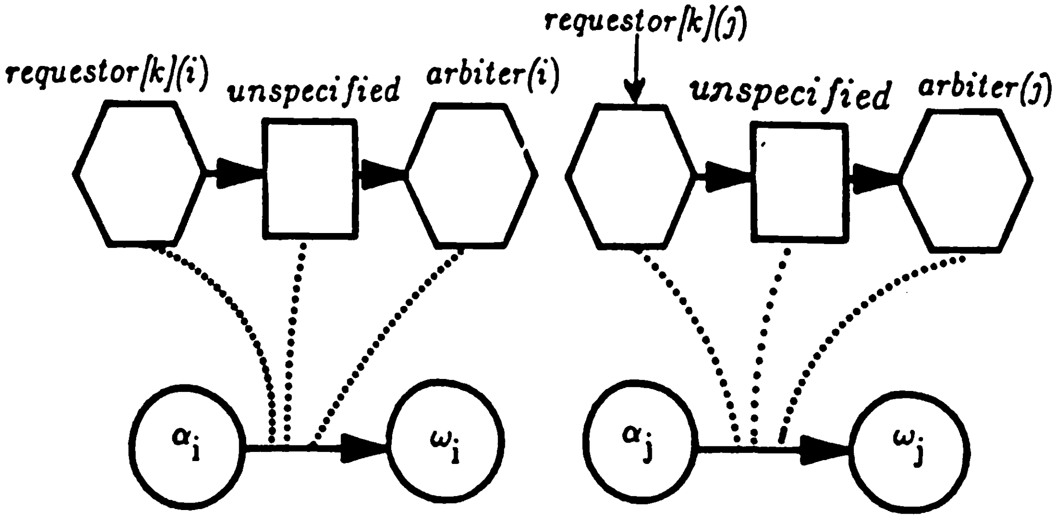


Figure 4-8: The fragments produced by PHRAN-SPAN for the BVT example sentence.

4.8. The Value-Carrier-Net-Range Binding

The **Value-Carrier-Net-Range** binding is another primitive in our SRL. It was created to capture the semantics of natural language constructs that map into a pair of DDS bindings, the **value-carrier-range** binding and the **carrier-net** binding.

4.8.1. The DDS template for a VCNR

The DDS representation of the value-carrier-range binding and the carrier net binding are identical to the DDS primitives described in Chapter 3.

4.8.2. The VCNR concept--a higher level of abstraction

As previously noted, natural language expressions tend to mix components from various DDS subspaces, *e.g.*, data flow and structural components. The notion of a binding in the DDS is a specific instance of the desire to relate components in different subspaces, *i.e.* interspace relationships. However, just as in the previous two concepts, the nature of a structural or physical component can usually not be determined at the single sentence level; hence, the **abstract component** is introduced. The result is that the two DDS primitives are combined into a single concept, *i.e.* a VCNR binding as shown in Figure 4-9.

4.8.3. Example sentences for the VCNR concept

The example sentence that introduced the bidirectional value transfer in the previous section (shown below for reference) contained an additional primitive concept often found in specifications, the VCNR concept. This concept is associated with the phrase **via two lines**.

Each requestor communicates with the arbiter via two lines, a request line and a grant line.

The VCNR concept is an extension of the value carrier range relation found in the DDS. The VCNR concept simply relates an abstract component to a data flow value and its range in the timing and sequencing subspace. This is consistent with matching the semantics of the concepts with the natural language representation. The concept for the example sentence is:

```
(v_c_n_r  (df_val *unspecified*)
          (a_component lines1)
          (ts_interval *unspecified*)).
```

The SRL primitive *ts_interval* introduced here corresponds to the **range**, (*i.e.*, time range or interval) in the VCNR concept.

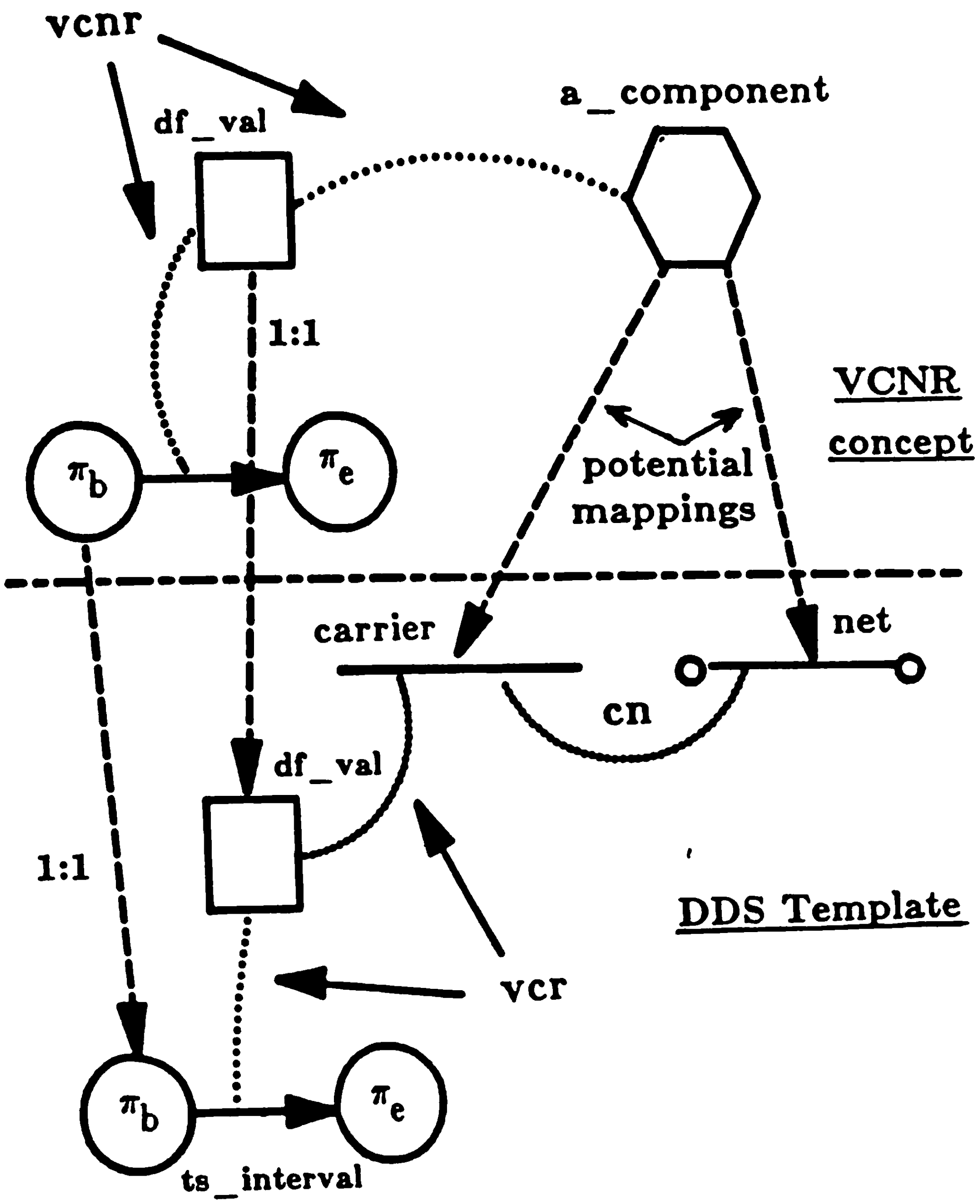


Figure 4-9: The VCNR concept and mappings into the DDS template.

When the example sentence is processed by PHRAN this concept is appended to the BVT concept resulting in the following output:

```
(bi_dir_vtrans (src/snk (a_component requestor1))
               (src/snk (a_component arbiter1))
               (info (df_val *unspecified*))
               (control (a_component *unspecified*))
               (v_c_n_r  (df_val *unspecified*)
                         (a_component lines1)
                         (ts_interval *unspecified*))).
```

In this example, the **info** is not specified in the BVT concept or the VCNR concept because it did not occur in the sentence; at other times, the presence in one slot or the other would permit the incompleteness in the specification to be corrected. Another example of a sentence containing the VCNR concept is

The peripheral equipment shall sample the !EF code word which is on the !OD lines.

The concepts output by PHRAN for this sentence are:

```
(v_c_n_r  (df_val !ef-code-word1)
          (a_component !od-lines1)
          (ts_interval *unspecified*))

(uni_dir_vtrans (sourc (a_component *unspecified*))
                (sink (a_component peripheral-equipment1))
                (info (df_val !ef-code-word1))
                (control (a_component *unspecified*))).
```

In this example the concept is introduced by a relative clause; this results in a similar VCNR concept but a slightly modified format which is ouput by PHRAN.

Instead of appending the VCNR concept directly to the value transfer as in the previous example the VCNR concept is added to a list of supplementary concepts that would be combined with the UVT or BVT concept in SPAN's post processing. Note also that the **info** is identified in the UVT concept as a data flow value and the **df_value** or data flow value is similarly identified in the VCNR concept.

4.8.3.1. Another level of semantics

The example sentence from the previous section (shown below for reference) seems to contain more information than the BVT concept captures.

Each requestor communicates with the arbiter via two lines, a request line and a grant line.

This additional information is associated with the semantics of individual words. A human reader of the specification might make use of these semantics to infer more about the value transfers being described. For example, the direction of the value transfers might be inferred from the word semantics: that a requestor makes requests which are placed on the request line and that the arbiter would produce grants which are placed on the grant line. The current version of PHRAN-SPAN cannot use the semantics of the individual words arbiter, requestor, grant and request to reason about the fragments of graphs in the DDS.

4.9. The Nondirectional Value Transfer

The abstract behavior of an NVT (Nondirectional Value Transfer) is *the input and output of data flow values to a single operation* in the data flow subspace and the associated timing and control information.

4.9.1. The DDS template for an NVT

The DDS template for the NVT consists of two subgraphs and their associated bindings as shown in Figure 4-10.

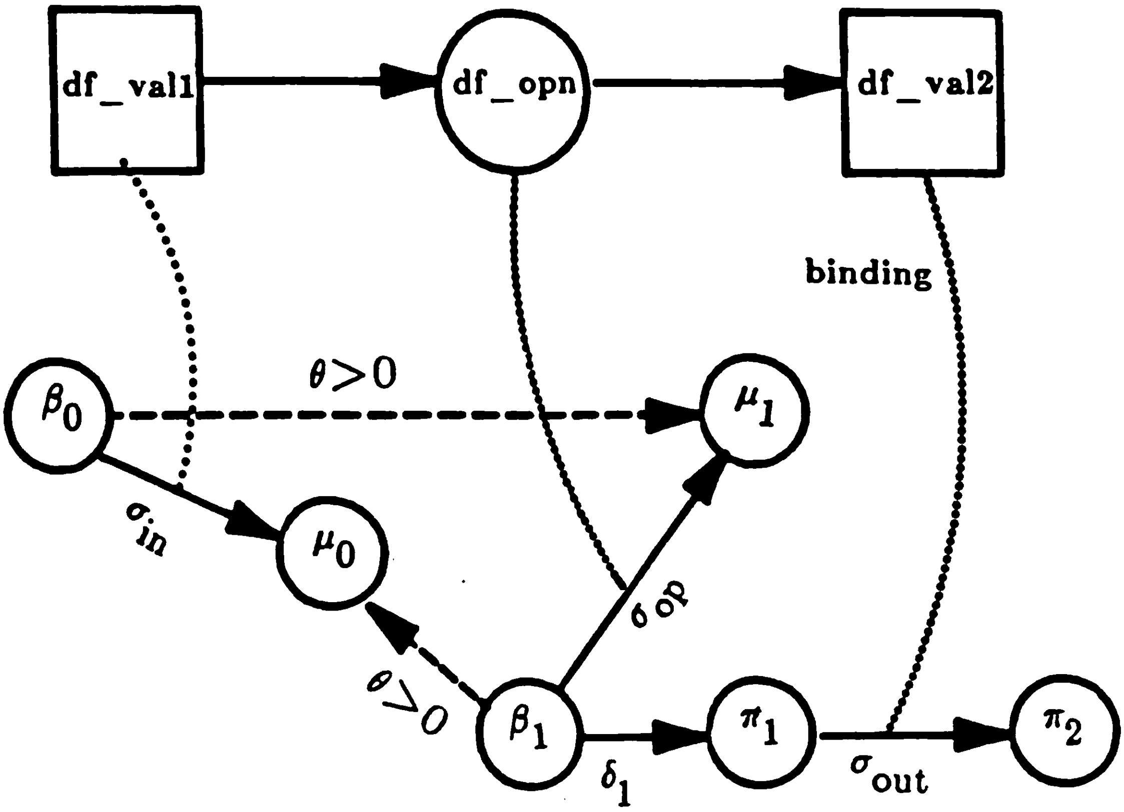


Figure 4-10: The DDS template for an NVT.

As with the UVT and the BVT, this representation defines the behavior associated with an NVT. There is no theoretical limit to the number of input or output data flow values associated with the data flow operation; however, PHRAN-SPAN curently restricts these to two, three or four depending on the particular NVT concept. Unlike the UVT and BVT, the control information is not explicitly represented in this representation but may be included as a value input to the data flow operation.

4.9.2. The NVT concept--a higher level of abstraction

The NVT concept is derived from the abstract behavior of an NVT and examples taken from natural language specifications. The NVT focuses on the input and output data flow values which are specified in the natural language construct. Like the UVT and the BVT, the NVT deals with the mixed behavior and structure present in natural language through the use of **abstract components**. The mapping of an NVT concept into the NVT DDS template is shown in Figure 4-11.

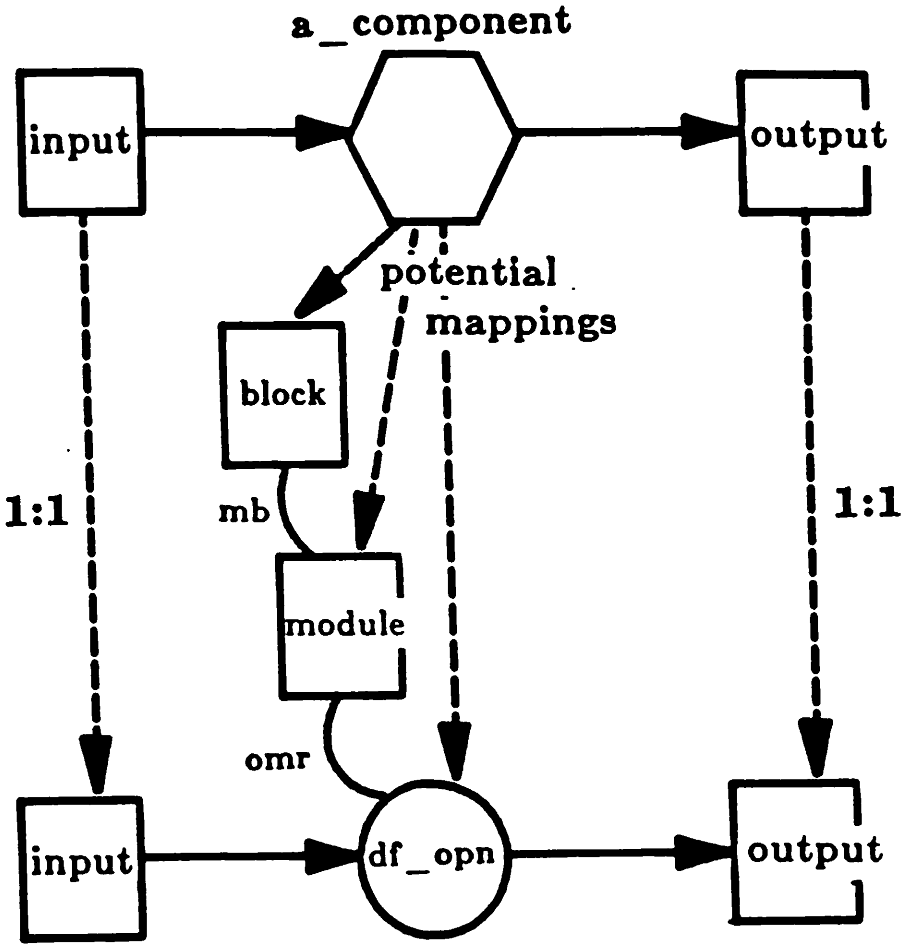


Figure 4-11: The NVT concept and mappings into the DDS template.

4.9.3. Example Sentences for the NVT concept

An example of a hypothetical NVT sentence that PHRAN-SPAN can analyze is

The cpu computes the difference of !a and !b.

For functions like subtraction and division where the inputs are not commutable, a simple convention is established that the inputs are read from left to right and the function inserted as for infix notation. Therefore the above sentence has the value **!b** subtracted from the value **!a**. Another approach could use appositives to identify the subtrahend, the divisor, *etc.* In addition to **compute**, other verbs that express the NVT concept are **count**, **form**, **keep**, **maintain**, **preserve** and **retain**. When used in a sentence, these verbs relate inputs and outputs to a single abstract component, resulting in an NVT concept:

```
(non_dir_vtrans  (fnc_info ?fnc_info)
                  (o_m_b_r (df_opn ?df_opn)
                           (a_component ?actor)
                           (ts_interval ?ts_interval)))
```

The new SRL primitive *o_m_b_r*, introduced here, signifies a **operation-module-block-range** binding or **OMBR**; like the VCNR binding this abstract binding replaces two DDS bindings the **operation-module-range** binding and the **module-block** binding. The SRL primitives used for the remaining DDS objects are the same as the VCNR binding in Section 4.8. Unlike the UVT and BVT the NVT concept is not fully self contained. The SRL primitive *fnc_info* determines the arity of the function and thus allows the NVT concept to have a data flow operation or abstract component with one, two or more inputs and also a variable number of outputs.

An example of this concept for a function with one output and two inputs is

```
`(fnc_info
    (input1 (a_value ,(new-token (a_value)(value 4 word)))
    (input2 (a_value ,(new-token '(a_value)(value 6 word))))
    (output1 (a_value *unspecified*))
    (operation (df_opn ,(new-token '(df_opn)(value 2 word)))
               (arity 2)))])
```

Also, the SRL primitive **a_value** that represents an abstract value is introduced here. The abstract value is necessary, because, at the sentence level, the inputs **!a** and **!b** in our example may be data flow values or structural carriers or physical nets. The **operation** part of this concept identifies the particular operation, in this case, the difference operation, and the **arity** of this function. The arity is specified to aid SPAN in analyzing structures with variable number of slots in the frames rather than by trial and error checking. Currently, the functions of sum and product accept up to four inputs; however, this is only an implementation limitation. Functions could also be defined with more than a single output. For example, the quotient could specify a remainder and an underflow. In the current implementation, it was assumed that SPAN could obtain this type of *implementation* information from the component library and request additional information from the specifier if necessary.

4.10. The Declaration Concept

The **declaration** is a concept created to capture the semantics of natural language sentences that reference data flow operations or data flow values, ranges of time or events, modules or carriers, and blocks or nets by a user-supplied name. The user-supplied names are differentiated from normal input in the current implementation by prefixing the *name* with an !, *e.g.*, cpu !a, !cpua or !disk1. These sentences establish the existence of objects in the DDS, associate DDS objects with library components, permit the user to differentiate among multiple occurences of a similar component and can sometimes remove the ambiguity associated with an *abstract component* or an *abstract value*. The SRL representation of a declaration is

```
(declaration
        (pname ?descriptor)
        (ref ?ref)
        (class ?class)
        (description ?nominative))
```

This concept links a user-defined *pname* with a class of DDS object, *e.g.*, the *pname* !a with the *class* of object cpu. All of the meanings of cpu would then be inherited by !a. This inheritance is accomplished in SPAN, where the declaration concept is recognized and an appropriate entry is generated in PHRAN's database. The SRL primitive *ref* determines whether the reference is definite or indefinite. By examining the type of reference, SPAN could decide to make a generic assignment to a class of objects or a specific assignment linking this object to a another named object or a specifically referenced object. Currently, SPAN only makes assignments to classes of objects, but could be easily extended. The *description* property would be used in conjuhction with the definite reference to a specific object.

4.10.1. The DDS template for a Declaration

In most cases, the DDS template for a declaration is simply a *named* object in the appropriate DDS subspace. More complicated cases generally can be mapped into an OMBR concept or a VCNR concept by SPAN.

4.10.2. Example Sentences for Declarations

Declarations as defined here do not often appear explicitly in actual natural language specifications but are implicit and usually must be inferred by the designer. However, the solution proposed in this thesis requires that there be a declaration section in each part of a specification to reduce the amount of implicit knowledge and make the specification more complete and less ambiguous. The following examples are hypothetical sentences that would be required by PHRAN-SPAN to process a natural language specification.

!a is a cpu.
!b is a cpu.

This example allows the user to reference either of two identical cpus; however, it does not allow us to disambiguate the use of cpu as a logical or physical component.

The next example references specific physical devices and therefore allows the named objects cpu !a and cpu !b to be associated with a **block** in the physical subspace.

The cpu !a is an IBM 370.
The cpu !b is a VAX 11/780.

In addition, if the physical block is a library component, its associated behavior from the data flow and timing and sequencing subspaces and its module(s) in the logical subspace can replace other references to the abstract component.

The last example is characteristic of a *global* decalaration and could be used by SPAN in post processing a group of sentences.

The following section describes the system's logical architecture.

This statement might be characterized as a meta-specification statement but is nevertheless classified as a **declaration** here and considered to aid in the disambiguation of a group of abstract component references. Since this meta-specification type of declaration involves modifying multiple sentences, it is not implemented in the prototype system.

4.11. Single Temporal Relation Concept

A STR (Single Temporal Relation) is a concept created to capture the semantics of natural language sentences that establish the partial ordering of various events in the timing and sequencing subspace. The events may be part of other concepts like a UVT, BVT, NVT or more complex temporal constructs, like the CTI (Causal Temporal **Inititiation** or the CTT (Causal Temporal Termination) that will be described in Section 4.13 and 4.14. Similar to the **declaration** concept, the STR concept does not represent a higher level of abstraction but maps directly into objects in the timing and sequencing subspace.

4.11.1. The DDS template for a STR concept

The DDS template for the STR concept is an arc in the timing and sequencing subspace and the events it orders. The arc's type, the events that are associated with its head and its tail, the length of the arc, the relation associated with the length of the arc and the units used for the length are all part of the DDS template for this concept, as shown in Figure 4-12.

4.11.2. Example Sentences for the STR concept

Temporal relations are often found in natural language specifications. The first example is

The computer shall clear the !ODA line before placing the next word on the !OD lines.

The adverbial phrase *before placing the next word on the !OD lines* is ordered with respect to the event of the !ODA line being cleared. This use of adverbial phrases is typical for this concept and examples of the beginning of adverbial phrases are: **prior to, 20 ns before, immediately following** and **after**. Two additional example sentences are

1. **!Select shall be dropped 100 ns after the write is begun.**

2. **After the exception lines go inactive the recovery phase begins.**

The SRL form of the concept that is used in the analysis of these sentences is

```
(single_temporal_rel
    (ts_arc_constraint  (arc_type ,(default '*constraint*))
                        (arc_head ,(default '*pred*))
                        (arc_tail ,(default '*succ*))
                        (arc_rel  ,(default 'gt))
                        (arc_len  ,(default 0))
                        (arc_units ,(default 'seconds))
    (*pred*  ,(value 2 cd-form))))])
```

In the example sentences used to demonstrate an STR, there are usually two or more concepts. For example, there is a concept associated with the main part of the sentence,

!Select shall be dropped.

Then there is the STR concept associated with the adverbial phrase,

100 ns after the write is begun.

The STR concept is appended to the concept associated with the main part of the sentence. Depending on the preposition used one of these concepts describes the predecessor, denoted *pred* in the STR frame and the other concept is the successor, denoted *succ* in the STR frame. Only the part of the concept that is appended onto the main concept is discussed here. The DDS template for this STR example is shown in Figure 4-12.

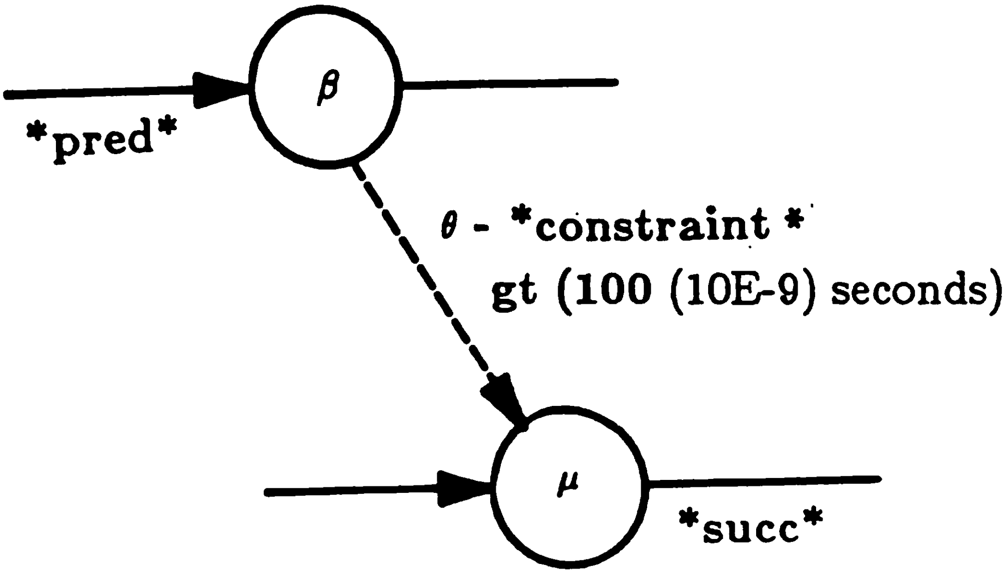


Figure 4-12: The DDS template for the STR concept generated by the example sentence.

As with previous concepts all the facet values of the STR concept are filled in by defaults; however, unspecified values aren't required as defaults, since the defaults are determined by the prepositions that are used in the adverbial phrase. The defaults for *arc_type* and *arc_rel* (arc relation) correspond to the arc types and relations as discussed in Chapter 3 on the DDS. The defaults ***pred*** and ***succ*** are the predecessor or preceding interval in time and the successor in time, respectively. Since the adverbial phrase that creates this concept modifies an existing concept by appending the STR concept to it, the slot values for the *pred* and *succ* are determined by the meaning of the particular adverb. The adverb **after** fills the *pred* facet slot with the part of the sentence that completes the adverbial phrase and the *succ* is the other concept, whereas for the adverb **before** the *succ* facet slot is filled and the *pred* is the other concept. Additional information (*e.g.*, the phrase *100 ns after*) would modify the default values in the facet slots for arc_rel, arc_len and arc_units resulting in the following concept for example #1:

```
(single_temporal_rel
   (ts_arc_1   (arc_type *constraint*))
               (arc_head *succ*))
               (arc_tail *pred*))
               (arc_rel  gt))
               (arc_len  100))
               (arc_units  1E-09 seconds)))
    (*pred*  (uni_dir_vtrans
               (source  (a__component *unspecified*))
               (sink    (a_component *unspecified*))
               (info    (df_val *high*))
               (control (a_component *unspecified*))
             (v_c_n_r
               (df_val *high*)
               (a_component !select)
               (ts_interval *unspecified*)))).
```

4.12. The Dual Temporal Relation Concept

A very important class of temporal constraints that are not described by the STR concept is the DTR (Dual Temporal Relation) concept. The prepositions **during** and **while** are typical examples of this class of temporal relations.

4.12.1. The DDS template for the DTR concept

The DDS template for the DTR is shown in Figure 4-13.

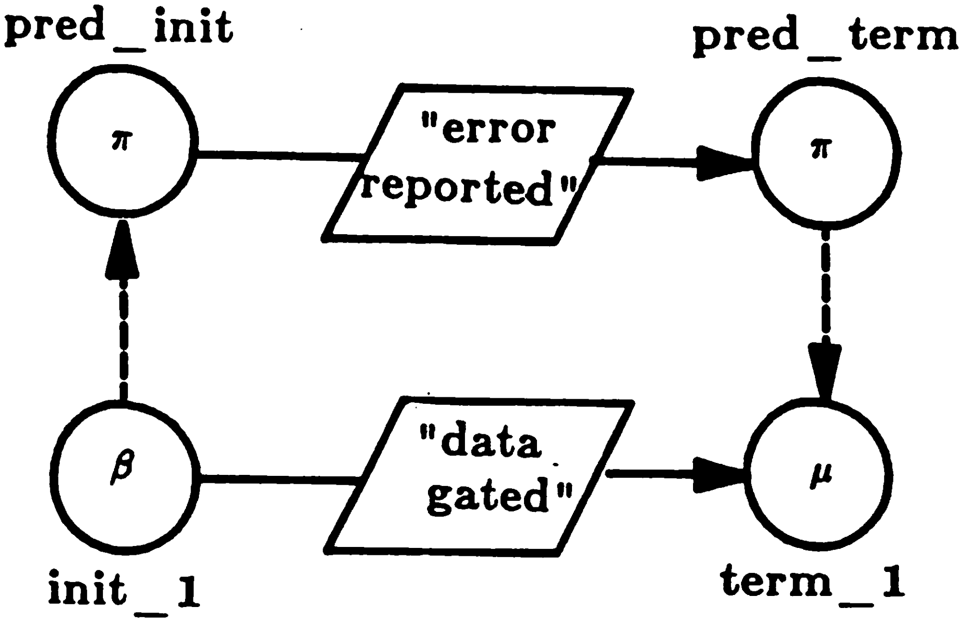


Figure 4-13: The DDS template for a DTR concept.

Because it unambiguously establishes the timing and sequencing relationship between two distinct intervals or ranges it consists of two constraint arcs which are referenced to the intitiation of both intervals, labelled init_1 and pred_init in the figure, and to the termination of both intervals labelled term_1 and pred_term.

4.12.2. Example sentences for the DTR concept

Two sentences expressing the DTR concept follow:

1. **The error will be reported on the !PS !Parity !Error line during the time the data is gated onto the !EC !Input !Data !Bus.**

2. **The inputs are not latched while the outputs are latched.**

The SRL representation of a DTR concept is

```
(dual_temporal_rel
     (ts_event_init_1 (event_name
                         ,(or (value 2) (value 2 word)
                              (default '*unspecified*)))
                      (event_type ,(default '*beta*)))
     (ts_event_term_1 (event_name
                         ,(or (value 2) (value 2 word)
                              (default '*unspecified*)))
                      (event_type ,(default '*mu*)))
     (ts_arc_1        (arc_typei ,(default '*constraint*))
                      (arc_tail ,(default '*beta*))
                      (arc_head ,(default '*pred-init*))
                      (arc_rel ,(default 'gt))
                      (arc_len ,(default 0)))
     (ts_arc_2        (arc_type ,(default '*constraint*))
                      (arc_tail ,(default '*pred-term*))
                      (arc_head  ,(default '*mu*))
                      (arc_rel ,(default 'gt))
                      (arc_len  ,(default 0))))))]))
```

The SRL primitives are the same as those used in the STR concept; however, there are more events and arcs. The two events *ts_event_init_1* and *ts_event_term_1* correspond to the initiating event and terminating event respectively of the interval that bounds the other interval temporally. That is if we say that an interval, A occurs during an interval, B we mean that A starts sometime after B starts and that A ends sometime before B ends, which means B bounds A. The default values for arc types and relations correspond to the values discussed in Chapter 3 on the DDS. The default values ***pred-init*** and ***pred-term*** refer to the initiation and termination of the concept preceding the DTR, *i.e.* the concept that the DTR is appended to by PHRAN.

4.13. The Causal Temporal Initiation

A hypothetical example sentence for a CTI is

The cpu starts the memory data transfer activity.

A CTI (Causal Temporal Initiation) extends the coverage of the STR and DTR concepts to allow direct reference to sequences of events and their relationships as a single concept. This often occurs in natural language specifications where relationships between subprocesses and process control are being described.

4.13.1. The DDS template for the CTI concept

The DDS template for a CTI consists of two subgraphs in the timing and sequencing subspace that are connected by a causal arc as shown in Figure 4-14.

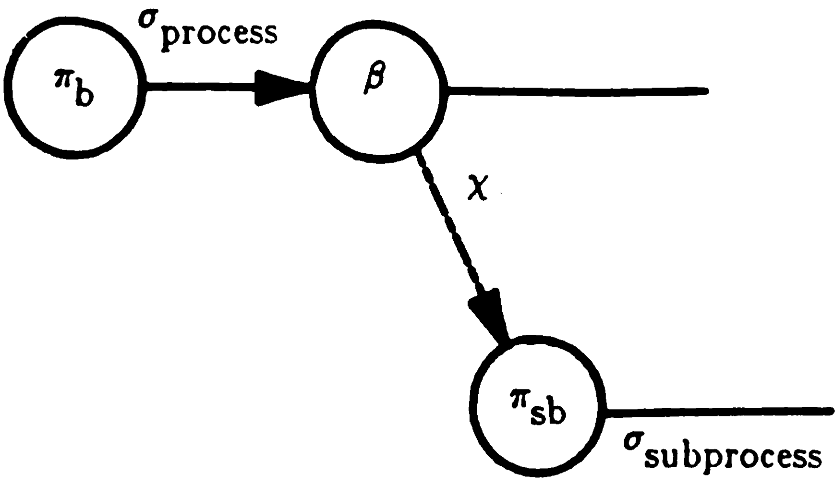


Figure 4-14: The DDS template for a CTI concept.

4.13.2. The CTI concept

The CTI *concept* focuses on the causal relation between two temporal series of events. Usually only the top-level details describing the events are included in the natural language sentences that correspond to the CTI concept. The CTI concept defines the controlling process' temporal activity by an interval in the TSss that starts with an event, *ts_event_init_1* which is followed by a second event, *ts_event_caus_1* that is a beta-type point and represents the initiation of the causal activity and the continuation of the main process. The causal event is linked in time through the causal arc with the start of the subprocess. This causal relationship can be seen in Figure 4-14.

4.13.3. Example sentences for the CTI concept

The verb **start** in the hypothetical example captures the semantic concept of causality. Two other verbs that express a similar semantic concept are **begin** and **initiate**. The complete CTI is expressed in the SRL as a frame-like data structure:

```
(causal_temporal_init
      (ts_event_init_1 (event_name ?actor)
                       (event_type ?event_type_init_1))
      (ts_arc_1        (arc_type ?arc_type_1)
                       (arc_tail ?actor)
                       (arc_head ?event_name_caus_1)
                       (arc_rel ?arc_rel_1)
                       (arc_len ?arc_len_1)
                       (arc_units ?arc_units_1))
      (ts_event_caus_1 (event_name ?event_name_caus_1)
                       (event_type ?event_type_caus_1))
      (ts_event_init_2 (event_name ?a_component_2)
                       (event_type ?event_type_init_2))
      (ts_arc_2        (arc_type ?arc_type_2)
                       (arc_tail ?arc_tail_2)
                       (arc_head ?a_component_2)
                       (arc_rel ?arc_rel_2)
                       (arc_len  ?arc_len_2)
                       (arc_units ?arc_units_2)))
```

The SRL primitives used here are explained in the description of the concept and in the description of the STR and DTR concepts.

4.13.3.1. The Single Temporal Event--a degenerate CTI

In certain cases, where there is no direct object in the sentence for a CTI concept, the complete concept can be described simply by the first event in the CTI, *ts_event_init_1*. Since the rest of the concept is not needed for this case a concept called a Single Temporal Event (STE) was created. An example sentence for the STE concept is

After the propagation delay, the data transfer starts.

This simple concept corresponds to the point in the TSss where the interval starts. The SRL form of the concept is

```
(single_temporal_event
          (ts_event_init_1 (event_name ?actor)
                           (event_type ?event_type)))

  actor  '?subject
  event_type_1 (default '*pi*)]).
```

4.14. The Causal Temporal Termination Concept

A hypothetical example sentence for a CTT is

The cpu terminates the print server.

The CTT (Causal Temporal Termination) concept like the CTI concept describes the causal relationship between a process and a subprocess. It differs from the CTI in the way the causal links are established between the process and the subprocess. In the CTT the causal link requires an asynchronous predicate to be defined with respect to the subprocess because the controlling process' "termination signal" is asynchronous with respect to the subprocess' clock.

4.14.1. The DDS template for the CTT concept

Like the CTI concept, the CTT concept consists of two subgraphs in the timing and sequencing subspaces. These subgraphs are linked through a bundle of causal arcs that correspond to the gamma points of the asynchronous predicate. Since the gamma points of an arc with an asynchronous predicate are not drawn for clarity, the bundle of causal arcs are normally omitted from a CTT DDS template also for clarity. These arcs are shown in Figure 4-15 to illustrate the causal relation in the CTT concept.

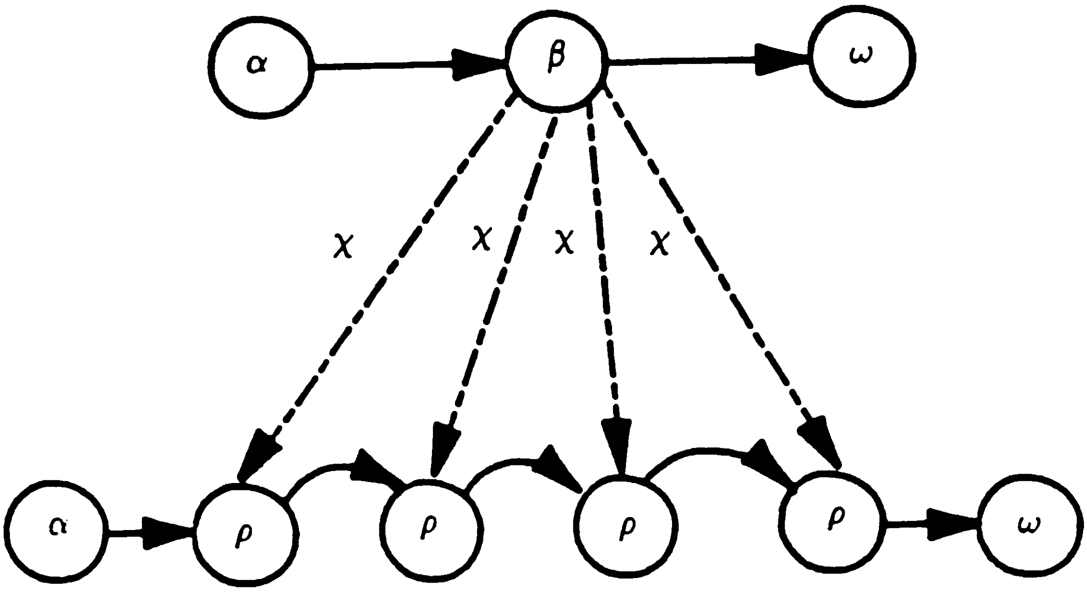


Figure 4-15: The DDS template for a CTT concept.

4.14.2. Example sentences for the CTT concept

The verb **terminate** in the hypothetical example captures the semantic concept of causality expressed in a CTT. Two other verbs that express a similar semantic concept are: **stop** and **end**. The CTT concept is expressed in the SRL as a frame-like data structure:

```
(causal_temporal_term
    (ts_event_init_1 (event_name ?actor)
                     (event_type ?event_type_init_1))
    (ts_arc_1        (arc_type ?arc_type_1)
                     (arc_tail ?actor)
                     (arc_head ?event_name_caus_1)
                     (arc_rel ?arc_rel_1)
                     (arc_len ?arc_len_1)
                     (arc_units ?arc_units_1))
   (ts_event_inter_1 (event_name ?actor)
                     (event_type ?event_type_inter_1))
    (ts_event_init_2 (event_name ?a_component_2)
                     (event_type ?event_type_init_2))
    (ts_arc_2        (arc_type ?arc_type_2)
                     (arc_tail ?a_component_2)
                     (arc_head ?a_component_2)
                     (arc_rel ?arc_rel_2)
                     (arc_len  ?arc_len_2)
                     (arc_units ?arc_units_2))
   (ts_event_inter_2 (event_name ?a_component_2)
                     (event_type ?event_type_inter_2))
    (asynch_pred     (predicate *unspecified*)
                     (a_component *unspecified*)
                     (ts_interval ?ts_event_init_1)
                     (ts_destination (event_name ?terminal)
                                      (event_type *rho*))))
```

The SRL primitives are basically the same as in the CTI concept except that there are more arcs and events and the asynchronous predicate has been added. The asynchronous predicate, *asynch_pred* consists of the Boolean expression, *predicate* which defines the condition which causes the asynchronous branch, the *a_component* that is the carrier required for an asynchronous predicate, the *ts_interval* that the predicate is bound to and the *ts_destination* that defines the rho node that the activity branches to when the predicate is true.

4.15. Asynchronous Temporal Activity

The ATA (Asynchronous Temporal Activity) is used in conjunction with other concepts like the UVT, BVT, NVT, CTI, CTT and STE. The ATA associates an asynchronous predicate with these concepts. The ATA is described by an adverbial clause and like the STR and DTR modifies another concept. An example sentence that demonstrates this is

Upon receipt of the flag, the cpu sends the data to the device.

The basic sentence is a UVT. This UVT occurs as a result of the condition associated with receiving the value associated with a certain flag. Another example is

When the data ready line is dropped, the device starts the data transfer process.

In this example, the basic sentence is a CTI concept, and the point at which the device starts the process is determined by the predicate produced by the ATA concept for the data ready line being dropped.

4.15.1. The DDS template for an ATA

The template for the ATA is simply the asynchronous predicate derived from the information contained in the adverbial phrase. The ATA concept is expressed in the SRL as a frame-like data structure:

```
(asynch_pred      (predicate  ?predicate)
                  (a_component ?a_component)
                  (ts_interval ?ts_interval)
                  (ts_destination (event_name ?terminal)
                                  (event_type *rho*))))
```

4.16. Summary

In this chapter we have defined a set of concepts that can be used to describe the behavior of digital systems. In building a prototype system for understanding natural language specifications, these concepts, along with lower-level concepts associated with the meaning of phrases have proven to be sufficient for a broad variety of sentences found in actual specifcations. Test cases run on the prototype system are presented in Chapter 6. Some of the lower-level concepts associated with phrases are discussed in the next chapter on natural language processing. Additional examples of all the concepts can be found in Appendix B.

Chapter 5

Natural Language Processing

5.1. Introduction

This chapter will discuss the natural language processing issues which arose in this research. It will begin by reviewing the components of the natural language interface. The theory and operation of PHRAN will then be summarized. Specific extensions and modifications to PHRAN for this problem will be described. Finally, a format for writing specifications in natural language will be outlined.

5.2. The Components of the Natural Language Interface

As discussed in the introduction to Chapter 4, the four components necessary to understand the specification of digital systems in restricted English text are

1. a corpus (a collection of writings, in this case examples) for the domain of these specifications,
2. a representation for the knowledge expressed in the corpus,
3. a formal representation for the behavior of a digital system, and
4. a parsing technique to map the natural language into the formal "behavioral" representation.

5.3. Examples of PHRAN operation

Having presented the requirements necessary for the natural language processing of a specification, some illustrative examples of the various representations and their use in processing the natural language input will be provided.

5.3.1. A simple sentence

A complete description of PHRAN and its operation can be found in [Arens 86]. The following description is adapted from one of the examples in Aren's thesis.

The cpu sends the data to the peripheral device.

PHRAN first reads the word *the* and the pattern-suggesting routine suggests patterns associated with the word *the*. PHRAN then forms a **term** for *the* and adds it to a list called ***PHRAN-BUF***. When the second word *cpu* is read, it matches the pattern consisting of the literal *cpu* and the concept associated with this pattern causes a term to be formed that represents a noun and a particular object, cpu. The pattern suggesting routine instructs PHRAN to consider the "basic" pattern associated with <article><noun>. As these patterns are considered, if there were a pattern that consisted of <article><noun>, a new term corresponding to a noun phrase would be created. Although, this is correct here, in other cases, where the word *cpu* was followed by another noun and formed a longer noun phrase, *e.g.*, *cpu register*, the match would be premature and the parse would fail. To avoid this, the pattern for <article><noun> is extended to <article><noun><not-noun next>. The pattern <not-noun next> allows PHRAN to look ahead at the next word following the cpu to see if it is part of a longer noun phrase. In this example, since the word is a verb, *i.e.* not a noun, the pattern matches and the concept associated with the pattern causes a new term to be formed that represents a noun phrase and a particular object **cpu1**. This new term replaces the two terms that were in *PHRAN-BUF* with a single term corresponding to the noun phrase *the cpu*.

Send is read next. It matches the literal *send* and an appropriate term is formed. The pattern suggesting routine suggests the "basic" pattern associated with the verb *to send*: There are two patterns for **send**:

```
[ (or (a_component) (df_opn)) (root send) (df_val)]

[ (or (a_component) (df_opn)) (root send)
                    (or (a_component) (df_opn)) (df_val)]
```

The disjunctive condition expressed in these patterns by **or** simply means that the subject can be either an *a_component* (abstract component) or a *df_opn* (data flow operation).

The initial condition of the pattern for the verb is found to be satisfied by the first term in *PHRAN-BUF* and this fact is stored under that term. Succeeding ones will be checked to see if this partial match continues. The term that was formed after *send* is now added to the list. *PHRAN-BUF* now contains

```
<[cpu1 - a_component, np],[send - verb]>
```

Next, *the data* is processed and results in a noun phrase that satisfies the last condition in the pattern for the verb *send*.

The first pattern for a sentence with the verb *send* is matched. All the terms in *PHRAN-BUF* are replaced with a single term corresponding to the concept associated with the matched pattern.

PHRAN continues processing the sentence and now reads the word **to** which it identifies as a preposition. The pattern suggesting routine suggests several adverbial phrases beginning with the preposition **to**.[11] When the adverbial phrase **to the peripheral device** is finally matched, PHRAN modifies the value associated with the **sink** to "peripheral device" in the concept associated with the verb **send**.

The concept part of the pattern-concept pair for **send** matched in processing the example sentence is

```
 [concept '(uni_dir_vtrans
                     (source (a_component ?from))
                     (sink   (a_component ?to))
                     (info   (df_val ?df_val))
                     (control (a_component ?actor)))
actor '?subject
df_val (value 3)
actor (default '*unspecified*)
to   (default '*unspecified*)
from (default '*unspecified*)
df_val (default '*unspecified*)]).
```

The prefix **?** indicates variables which are associated with the facet values. For example, during processing **?df_val** is replaced by the **value** associated with the third element of the pattern, *the data* and the **?actor** is replaced by the value associated with the subject of the sentence, *the cpu*. The values are usually **tokens** created by PHRAN when the words are encountered. For example a token is created for *the cpu*, **cpu1** and tokens are also created for *the data*, **data1** and *the peripheral device*, **peripheral-device1**. Note, if the phrase *the cpu* or *the data* had been used in a previous sentence(s), then the token would have been **cpu2** or **data3** or some other numbered token. When the sentence is completely

---

[11] In the documentation on PHRAN, patterns of this nature are generally handled by making the adverbial phrase an optional part of the pattern for the verb **send**. The processing of optional phrases, although very efficient, is also rather restrictive. The optional patterns could not handle all of the constructions required for processing many sentences taken from specifications; therefore, in this implementation, adverbial phrases are always handled by a separate pattern-concept pair.

matched to patterns in PHRAN's database a term representing the meaning of the sentence is all that remains in *PHRAN-BUF* and all variables are replaced by a token or a default value. The result for processing this example is

```
(uni_dir_vtrans
    (source (a_component *unspecified*))
    (sink   (a_component peripheral-device1))
    (info   (df_val data1))
    (control (a_component cpu1)))
```

Given the tokens for the source operation, the sink operation, the control operation and the value transferred, the DDS template associated with the UVT concept can be filled-in, resulting in a data flow graph associated with the example sentence. In this example, no timing is specified so the canonical timing and sequencing subgraph is assigned by default.

Note that by not mapping directly from the English to the graphical representation, the knowledge is stored in a form easy for SPAN, the SPecification ANalysis program, to process for incomplete information. The difficulties associated with mapping and many level representations are cited as an open problem by [Weischedel 83].

5.3.2. Adding timing information

The addition of a single phrase to the end of the example sentence used in the previous section demonstrates the capability of this technique to handle timing information. We simply change the sentence to

The cpu sends the data to the peripheral device in less than 100 ns.

The information *in less than 100 ns* is easily handled as an adverbial phrase in PHRAN which modifies the basic UVT concept that it created when parsing the first part of the sentence. The first part of the sentence is processed and the basic UVT concept is formed, then the pattern associated with the phrase **in less than 100ns** is matched. This second match results in PHRAN testing the previous concept. When it finds uni_dir_vtrans, the concept is modified by appending the timing information extracted from the adverbial phrase to the existing UVT. The timing information is described by an STR, which was described in Section 4.11.

```
(ts_arc_constraint  (arc_type ,(default '*constraint*))
                    (arc_head ,(default '*pred*))
                    (arc_tail ,(default '*succ*))
                    (arc_rel  ,(default 'gt))
                    (arc_len  ,(default 0))
                    (arc_units ,(default 'seconds)))
(*pred*  ,(value 2 cd-form))))])
```

The modified concept indicates that there is a timing interval, that it has a **constraint** type arc associated with it and that the relation associated with the constraint arc is **lt**, for less than, the value is 100 and the units are nano-secs. At this level of abstraction, no additional information is available. The data flow value and the abstract components are all bound to the entire ts_interval. The DDS representation which incorporates the timing information is shown in Figure 5-1.

5.4. Extensions to PHRAN

Most of the extensions to PHRAN required to process specifications were made by augmenting PHRAN's knowledge base with generalized pattern-concept pairs. A few cases could not be accommodated by adding additional pattern-concept pairs but required small modifications to PHRAN's routines that performed the analysis; however, they simply added capabilities and did not interfere with existing processing. These small changes significantly enhanced PHRAN's basic capabilities for the domain

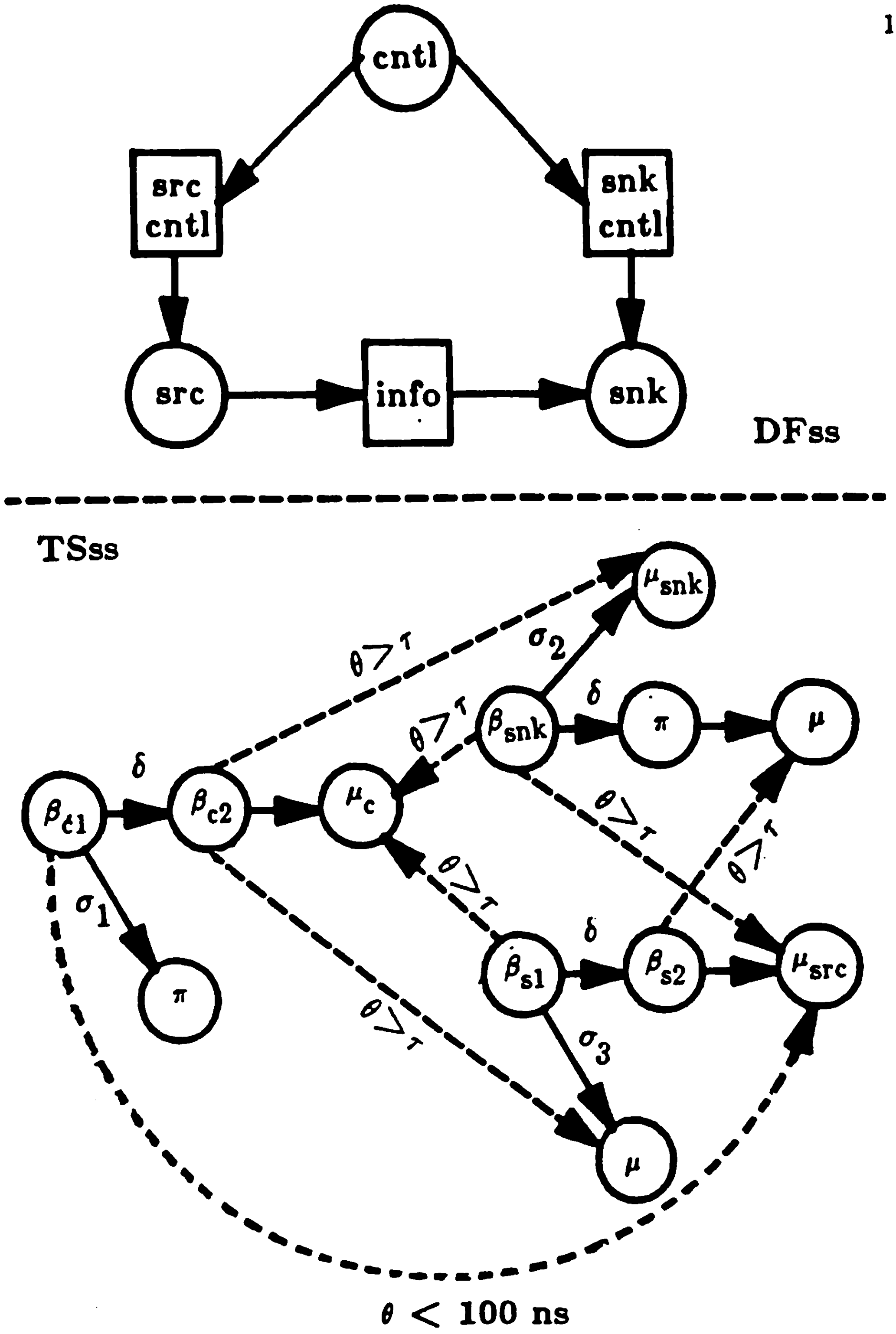


Figure 5-1: DDS representation of data flow and timing information.

of specification understanding. In the following sections, we will describe the handling of noun phrases, temporal and relational modifiers, numbers and naming.

5.4.1. Noun phrases

It is quite common for nouns to be used as modifiers of other nouns in specifications. Some examples are

1. *bus request cycle*,
2. *transfer block size*,
3. *segment trap request*,
4. *data transfer operations*,
5. *segment descriptor number*,
6. *data transfer bus cycles*,
7. *interrupt vector transfer phase*, and
8. *arithmetic register reference instruction*.

These phrases are often created by the specifier to reference a particular entity, *e.g.*, a piece of hardware, an activity, or a range of time. Therefore, their meaning can usually be inferred from the last noun in the phrase. However, the process of forming these groups of nouns into a specific noun phrase is complicated by the fact that many of the words used in specifications are syntactically ambiguous, *i.e.*, the word may be either a noun or a verb [Pavlovic-Lazetic 86]. Examples of these words are **interrupt**, **process**, **signal**, **start** and **transfer**. For example, if transfer is stored as a noun and as a verb, PHRAN will prefer the pattern suggested first, all other things being equal. Thus it would parse the *the process' transfers the data* incorrectly as <noun phrase: the process transfers>

<noun phrase: the data>. However, it would parse *the data transfer register* correctly, producing <noun phrase: the data transfer register>.

Therefore, PHRAN's inherent priority scheme, will result in a word, that can be used as either a noun or a verb, being recognized as a noun if the noun is indexed before the verb. PHRAN's access routine was modified to look up all possible meanings of a word; therefore, the only problem left to solve was when to use the word as a noun or as a verb. This is explained in the following section on disambiguation.

5.4.1.1. Rules for disambiguation

Some potential rules for resolving the word's use in a sentence are

1. check the agreement in number of the subject (potential noun phrase) with the verb/noun,
2. check whether the word preceding the verb/noun is an *active agent*, *i.e.* a possible subject of the sentence and/or
3. check whether the word following the verb/noun is a verb or a noun or another verb/noun.

Evaluation of several examples led to a simple heuristic based on rule #1 and rule #3. This heuristic also required two new functions to be added to PHRAN. The two functions added to PHRAN are

1. **noun-and-verb** that tests whether a word can be used as both a noun and a verb, and
2. **after-next** that looks ahead two words to the right.

The modification to PHRAN to look ahead further or back up a little more is consistent with other research on parsing (*e.g.*, PARSIFAL [Marcus 80] has a three-place constituent buffer). Rule #2 is also useful for resolving certain ambiguous cases but requires additional semantic information to be encoded at the word level to indicate potential active agents, whereas, the heuristic based on rule #1 and rule #3 is encoded in a small number of generalized pattern-concept pairs that handle noun phrases.

5.4.1.2. The heuristic for noun phrases

The patterns in PHRAN that are used to differentiate ambiguous noun phrases from those that are unambiguous are given below:

```
;The next pattern actually handles special noun noun cases but
;must be indexed under noun, to allow look ahead--the result
;is an ambiguous sentence.

(index-under-pattern (noun)
 [(nil
  [ (or (p-o-s article) (p-o-s quantifier)); Heuristic 2
    (* (and (p-o-s noun)                    ; a noun followed
                (noun-and-verb next)        ; by a noun-and-verb
                (not (not-plural next))     ; singular in number
                (noun-and-verb after-next)))
                                          ; followed by a second
     (p-o-s noun) (p-o-s noun)]             ; noun-and-verb
                                            ;
 [ p-o-s 'ambiguous-sentence
   do (add-to-*sc* '(ambiguous-sentence (verb1 ,(value 3 word))
                                   (verb2 ,(value 4 word)))))]))


;The next pattern actually handles special noun noun cases but
;must be indexed under noun, to allow look ahead--the result
;is a noun phrase that contains one noun.

(index-under-pattern (noun)
 [(nil
  [ (or (p-o-s article) (p-o-s quantifier))      ; Heuristic 1
    (* (and (p-o-s noun)  (noun-and-verb next)   ; a noun next
                          (not(not-plural next)  ; and a verb
                 (not-verb after-next 'basic)))] ; with the
                                                 ; same stem
                                                 ; and plural
                                                 ; in number
                                                 ; and there isn't
                                                 ; a verb after
                                                 ; this noun
    [ p-o-s 'noun-phrase
      cd-form (old-token (value 2 description) (value 2 word))
      description (value 2 description)
      do (add-adjs-to-*sc* (value 1 adjs)
                           (terms cd-form))
      do (copy-term 3)])])
```

The *and* in these heuristic patterns indicate that all the conditions must be true for the pattern to match. The basic structure of the patterns for noun groups is similar to Winograd's **Noun Groups** [Winograd 72] and these heuristics are consistent with Gershman's rules used in the Noun Group Parser (NGP) [Gershman 79]. Our work differs from Winograd's research in that he did not address the specific syntactic ambiguity problem in forming the noun phrases, and from Gershman's work in that our patterns are more general and are not for specific classes of noun groups. (Even though these patterns are more general, they are not used in the formation of all noun phrases. For example, Gershman's patterns involving time are handled by a special class of patterns.)

The following examples will demonstrate the ability of this heuristic to detect this type of ambiguity in forming a noun phrase.

The input to PHRAN is

the cpu signals interrupt transfer activity.

This may be parsed as

1. <the cpu> <signals> <interrupt transfer activity>, or
2. <the cpu signals> <interrupt> <transfer activity>

When the example is parsed by PHRAN, the first two words are parsed as a potential noun phrase, then the word **signals** is encountered. The word **signals** can be either a plural noun or a singular verb (i.e., it can be used as a noun and also as a verb). The final condition is that the word after **signals** has a use as a noun and as a verb. **Interrupt** satisfies the final condition resulting in an **ambiguous-sentence** and SPAN informs the user of the possible meanings.

The ambiguity can often be removed by rewriting the sentence, for example, *the cpu shall signal interrupt transfer activity* is not ambiguous and corresponds to parse #1. Here the addition of the modal verb **shall** has removed the ambiguity. An unambiguous sentence resulting in parse #2 can be formed by inserting an article or quantifier between interrupt and transfer activity, for example, *the cpu signals interrupt all transfer activity*. A modal verb may also be used to remove the ambiguity in the case of parse #2, for example, *the cpu signals shall interrupt transfer activity*.

Having solved this ambiguity detection problem, the second part of the problem is to determine the meaning of the unambiguous noun phrases. The solution for this was based on the typical use of these complex noun phrases in specification documents. In general, these noun phrases are used to reference specific items in the specification, for example, *the data transfer register* refers to a specific *register*. Therefore, the nouns are simply concatenated together to form a unique token and the string is assigned the meaning of the last item. For the example, the token would be the **data-transfer-register1** and its meaning would be an **a_component**.

5.4.2. Quantitative information and numbers

In specifications, quantitative data is usually present. Understanding this data requires a knowledge of magnitude and units. PHRAN originally understood decimal numbers that were typed in as figures, *e.g.*, 97, 865 and 111111.

PHRAN's knowledge base has been extended to understand some numbers that are written as well (*e.g.*, one, forty two, three million).[12] PHRAN's knowledge about numbers also includes ordinal numbers as well as decimal fractions (*e.g.*, tenths, hundredths).

In specifying behavior, time is the principal quantitative information that PHRAN processes. Basic patterns were created that allow PHRAN to form the meaning of temporal quantities, capturing the magnitude and units and any qualifying relations (*e.g.*, less than, more than, not more than).

The basic units of time are seconds and PHRAN's knowledge base has been extended to handle nanoseconds, microseconds, milliseconds and seconds, uniformly. In addition, PHRAN recognizes the common abbreviations associated with each of these units, *e.g.* ns, nanosecs or nanoseconds. PHRAN can also translate minutes and hours into seconds.

An example of the type of temporal phrase PHRAN encounters is

less than 24 ns.

PHRAN would first read the word *less* and recognize it to be an adjective; next PHRAN reads the word *than* and matches the phrase *less than*. The pattern-concept pair for this phrase is

```
[ less (* and than) ]
[p-o-s 'adv-rel
 description '(relation)
 cd-form 'lt ])]).
```

Next *24* is read and PHRAN associates this with the concept of a number. Finally, *ns*, the abbreviation for nanoseconds is read. The pattern for *ns*

---

[12] Manuals on style [Perrin 59], [McCrimmon 63] recommend that only numbers that can be expressed in one word be written out and that figures be used otherwise.

references the basic pattern for nanoseconds resulting in the following concept:

```
cd-form  1E-9
description '(ts_units)
```

Now that the three primitive concepts have been formed PHRAN matches the pattern

```
(relation number ts_units)
```

and completes the concept for the phrase *less than 24 ns*. The completed concept is

```
( ts_measure (relation lt)(amount 2.4E-08)
                          (units seconds)).
```

This concept called a ts_measure is then used in forming larger concepts dealing with temporal descriptions.

5.4.3. Named objects

The first thing most parsers do is to check if the word it has scanned is in its lexicon. PHRAN originally did this and then queried the user to check for possible misspelling or mistyping. If the user indicated that the word was correct as seen by PHRAN then PHRAN added the word to its lexicon without storing a *meaning* for the word. This represented a problem for processing specifications since users frequently name a device, *e.g.*, cpu A or a signal or line, *e.g.*, the EF code word or the OD lines. A simple convention of prefixing an exclamation point to these names allows PHRAN to trap these names and assign them the special class of a **pname**, *i.e.* a proper name for the object. For the previous examples the user would enter cpu !A, !EF code word and !OD lines. Each pname can be given the properties of the object that it names in a subsequent postprocessing phase like SPAN. This is accomplished explicitly with the concept of a declaration as introduced in Chapter 4.

5.5. Format of the Specification

Application of natural language processing to the specification problem can be greatly facilitated by structuring the specification document. The **IEEE Guide to Software Requirements Specifications** [IEEE 84] has been selected as the basis for a model of a specification document. In particular, we are concerned with the section of the specification document identified as the Functional Requirements.

Our proposed format is closest to Outline 3 for System Requirements Specification on page 24 of ANSI/IEEE Std. 830-1984. The system is described as a group of functional requirements. In each group of requirements, the introduction section is replaced by a set of declarations that define the objects referenced in the specification. Examples of this type of decalaration might be

- A process is a data flow operation.
- The ALU is a logical unit.
- All references to peripheral equipment are to the physical devices.

Next the inputs of the system are defined. Definition of the inputs should include

1. the source of the input,
2. the quantity,
3. the units of measure,
4. the timing characteristics, and
5. the range of the valid inputs including accuracies and tolerances.

Examples of each of these types of statements are

- **The inputs !a, !b and !c are external inputs to the system.**
- **The inputs !a, !b and !c arrive as a group.**
- **The inputs !a and !b are 32 bit words.**
- **The inputs shall be available for 100 ns.**
- **The inputs !a, !b and !c range from 0 to 255.**

The next section is the processing section. This section should define all of the operations to be performed on the input data including intermediate values that must be generated to obtain the output. It should include:

1. all the operations to be performed,
2. the exact sequence of operations, and
3. any response to abnormal situations.

For example:

- **The system reads the input values !a, !b and !c.**
- **The alu computes the sum of !a and !b.**
- **The cpu sends the result to the terminal.**
- **When the interrupt flag is raised, all memory transfers stop.**

In general, processing should be described in blocks of related operations. If there is a loop or iteration, it should be described explicitly. The place where the loop begins should be indicated by a statement such as

The loop !compute begins.

This statement would be followed by the body of the specification describing the activity in the loop. After all the processing was described, a statement such as

The loop !compute ends.

should be included. This approach allows us to avoid the hard problem of deciding where loops begin and end. We can then apply standard compiler technology to this problem.

In the last section, the outputs should be defined in the same manner as the inputs.

Chapter 6

Prototype System and Test Cases

6.1. Introduction

In this chapter we will discuss the prototype PHRAN-SPAN system constructed to validate the representations and methodology proposed in this research. The primary components of the prototype system are

1. the phrasal analyzer, PHRAN,
2. the specification analyzer, SPAN and
3. the knowledge base for PHRAN.

The knowledge base required by PHRAN for processing digital system specifications will be discussed in Section 6.2. In Section 6.3 we will describe the SRL form of PHRAN's output. Following that we discuss the results of processing several sentences taken from actual specifications. PHRAN has been described in Sections 4.4 and 5.3. The details of PHRAN operation can be found in [Arens 86]. SPAN's function is currently limited to recognizing the various concepts and reformatting PHRAN's output into English. Examples of SPAN's current capabilites will be shown in Section 6.4. The ultimate capabilites of SPAN will be discussed in Chapter 7.

6.2. The knowledge base

PHRAN's knowledge base consists of pattern-concept pairs which were described in Section 5.3.1. An example of the input used to create the knowledge base in the PHRAN-SPAN prototype can be found in Appendix B. The pattern-concept pairs may be stored in a variety of ways. We will first examine those that are stored as words corresponding to a particular part of speech, the most prevalent ones being nouns and verbs.

6.2.1. The nouns

Nouns are stored or indexed[13] in PHRAN's knowledge base by using the function **noun:**. For example, the noun *activity* would be indexed in PHRAN's knowledge base as follows:

```
(noun: activity activities (activity df_opn))
```

The arguments to the function **noun:** are the word, its plural form and a list of concepts, *i.e.*, the semantic categories that might be associated with the word. In this case, the word activity belongs to two categories, one which indicates that the word itself represents a semantic category, as well as being a word and the second entry assigns it to the semantic category *df_opn*, *i.e.*, a data flow operation. In addition to the category **df_opn**, there are ten other semantic categories that have been created for nouns to aid in the understanding of digital system specifications. Three of these categories correspond to objects in the DDS, they are **df_val**, **ts_event** and **ts_interval**. The **df_val** corresponds to a data flow value in the data flow subspace of the DDS. The **ts_event** corresponds to events or points in the timing and sequencing subspace (TSss) and the **ts_interval**

---

[13] Stored and indexed are used interchangeably in this thesis; however, index has a special meaning and the interested reader should consult the PHRAN documentation.

corresponds to ranges or intervals in the TSss. The **a_component** or abstract component and the **a_value** or abstract value form another category. This category was created to cover the uncertainty associated with the usage of a word (*e.g.*, the word cpu might refer to a logical unit, *i.e.*, a module in the structural subspace, or to a particular piece of hardware, *i.e.*, a block in the physical subspace). Similarly, reference to a signal by name may refer to the logical carrier or the physical net or possibly even a data flow value. Other examples of the use of this category are given in Chapter 4. Another semantic category was necessary for dealing with structured objects such as blocks, strings, records, stacks, and tables. This category was named **struct_obj**. The last semantic category is related to describing functions. This category was created to handle functions with different numbers of inputs uniformly. Currently there are three members of this category a **unary_fnc**, a **binary_fnc** and an **nary_fnc** for functions with one, two or more than two inputs, respectively.

There is some art to deciding which semantic categories to assign to a noun. Assigning more categories can impact processing time since a token can be created for each semantic category and when doing pattern matching each of the categories might have to be tried individually. The usage of the word must also be considered--some words may have many different uses. An example of this is the noun *interrupt* which has the following entry:

(noun: interrupt (interrupt temporary-halt break df_opn ts_event df_val a_value))

The noun interrupt is assigned to seven semantic categories. The first being the word itself, this represents a very specific usage, *i.e.*, the concept of *interrupt* would actually have to occur in the PHRAN pattern for this meaning to match. The second semantic category of a temporary-halt might match a less specific pattern in PHRAN, *e.g.*, one associated with the word *suspension*. The last four semantic categories are more general and would allow interrupt to match PHRAN's pattern in a large number of cases. Now that patterns to handle nouns and verbs with the same lexical stem have been added to PHRAN and PHRAN's knowledge base, these words may be stored like any other noun or verb. However, the entry for the verb must follow the entry for the noun.

Though over 2000 words have been included in the vocabulary list only about 100 of the nouns are indexed in the prototype system's knowledge base. These nouns are shown in Table 6-1 and the patterns used to index them can be found in Appendix B.

6.2.2. The verbs

Like nouns, verbs maybe indexed in PHRAN's knowledge base by using a special function **verb:**; however, this function only creates pattern-concept pairs that correspond to the various forms of the verb. For example, *send* and *transfer* would be indexed in their various forms by including the following line in the LISP file used as PHRAN's knowledge base.

```
(verb: send sent sending)
(verb: transfer transferred transferring)
```

Irregular verb forms can be indexed with this function by supplying four basic forms of the verb instead of three as indicated here. Also, in the exceptional case when the verb has unusual forms, they must be defined and indexed explicitly [Wilensky 80]. All the verbs used to date in the prototype system's knowledge base are regular (Appendix B).

access
acquisition
activity
address
agent
alu
arbiter
arbitration
arbitrator
bit
block
byte
bus
cache
call
channel
check
clock
code
command
condition
control
cosine
cotangent
count
cpu
cycle
data
delay
device
difference
end
equipment
event
fetch
fft
filter
flag
format
function
grant
handshake
index
information
input
interrupt
interval
line
map
mark
marker
match
measure
memory memories
multiplexer
operation
output
parity
pause
peripheral
phase
point
preset
priority
process
product
propagation
quotient
read
receipt
receiver
record
register
report
request
requestor
reset
resolution
resource
root
sender
sequence
shift
shifter
signal
sine
sort
speed
square
stack
start
store
switch
subsystem
sum
system
tag
tangent
task
test
transfer
trap
unit
use
value
wire
word
write

Table 6-1: Nouns in the prototype PHRAN-SPAN knowledge base.

The function for indexing verbs, unlike the one for indexing nouns, only forms patterns that cover the various forms of usage. It does not index the meaning of the verb or an actual pattern-concept pair. Therefore, the verb's meanings must be entered separately in the knowledge base. The **pattern-concept pair** associated with a verb is indexed with another special function **name**. For example, the following entry would be included in the knowledge base to index the pattern-concept pair for the verb *transfer*.

```
(name %transfer
  ((active passive)
   [(or (a_component) (df_opn)) (root transfer) (df_val)]
   [concept '(uni_dir_vtrans
                  (source (a_component ?from))
                  (sink   (a_component ?to))
                  (info   (df_val ?df_val))
                  (control (a_component ?actor)))
     actor  '?subject
     df_val (value 3)
     to     (default '*unspecified*)
     from   (default '*unspecified*)
     df_val (default '*unspecified*)]))
```

The first argument to the function **name** in this example is the root of the verb prefixed with a percent sign. This allows all forms of the verb defined by the function **verb:** to use this pattern. The next argument to **name** is a list containing several arguments. The first argument in the list identifies whether this pattern applies to the active voice, passive voice or both uses of the verb. The next entry in square brackets is the pattern part of the pattern-concept pair. The pattern here has three parts. The first part

```
(or (a_component) (df_opn))
```

indicates that the subject of the sentence formed by matching this pattern belongs either to the semantic category of **a_component** or to the semantic category of **df_opn**. The next part of the pattern

```
(root transfer)
```

indicates that this is the verb which has the root *transfer*. The last part of the pattern

```
(df_val)
```

indicates that the object being transferred belongs to the semantic category of data flow values. The last argument in the list is the concept **uni_dir_vtrans** associated with the pattern for *transfer*. This concept called a unidirectional value transfer is described in detail in Chapter 4 along with all the other concepts that PHRAN uses in analyzing digital system specifications.

Unlike nouns, where the word itself is the pattern, patterns must be created for each verb that cover all the possible uses of the verb. This requires that special patterns for only the active voice be separated from the patterns that are used to cover both the active and passive voice. In addition, verbs can require one or more adverbial phrases or patterns with direct or indirect objects to support their use. The basic patterns needed to support a UVT verb like **send** or **transfer** are discussed in Section 6.4.2.

There are 25 verbs whose meanings are currently stored in the prototype system's knowledge base.

6.2.3. Adverbial Phrases

Adverbial phrases are often used with verbs like **send**, **transfer**, and **communicate**. These phrases are indexed by the preposition used in the phrase. Schematic representations[14] of two sentences that contain adverbial phrases follow:

```
<component><sends><value><to><component>

<component><communicates><with><component>.
```

PHRAN's knowledge base contains adverbial phrases that use the following prepositions:

```
from    to    by    before    after    via    during   in
```

6.2.4. Numbers

In addition to the nouns, verbs and prepositions described in the previous subsections, the system's knowledge base contains patterns for the cardinal numbers[15] from one through twenty as well as the numbers for each decade, *i.e* thirty, forty, etc. The system's knowledge base also has patterns for the numbers a thousand, a million and a billion. The knowledge base also contains patterns that match decimal fractions such as tenths, hundredths, thousandths, millionths and billionths. Also, the knowledge base has patterns for the first ten ordinal numbers.

---

[14] These schematic representations use angle brackets to delimit the elements of a sentence. The elements may refer to generic concepts like *component* or *value*, specific words such as *send* and *to* in this example, or parts of speech like *noun* or *verb*.

[15] PHRAN has a basic pattern which matches any string of digits and converts it to a number internally.

6.2.5. Units

Though PHRAN's morphological routines could be set up to handle the metric prefixes like *kilo*, *mega*, *giga*, *milli*, *micro*, *nano*, *pico* and *femto*, these prefixes are handled in context by using separate patterns for each use of the prefix with a word. This also facilitates the use of abbreviations for various concepts.

6.2.5.1. Time

For a measure of time like nanoseconds, the patterns ns, nanosecs and nanoseconds are stored. A user could also specify a quantity of time by using the cardinal number with the basic unit of measurement *seconds* or its abbreviation *sec*. The patterns for millisec, microsec, nanosec, picosec, and femtosec and their variants are stored in the prototype system.

6.2.5.2. Storage and data rates

The most frequent uses of the prefixes kilo, mega and giga are to describe the quantity of storage required or the capacity of a channel to transfer information. Therefore, the abbreviations KB, MB and GB are stored along with the patterns for kilobytes, megabytes and gigabytes to describe storage capacity. To avoid notational confusion and misinterpretation the patterns describing data rates are stored as complete phrases. For example, *kilobits per second* or *megabits per sec* are typical patterns stored in the prototype system's knowledge base.

6.2.6. Determiners and qualifiers

Pattern-concept pairs for the determiners *a*, *an*, *the* and *each* are all indexed in the prototype system's knowledge base.

6.2.7. Extending the knowledge base

In general, adding more nouns to the knowledge base is straightforward and will probably produce no side effects. However, for other more complex patterns, specifically, general or lengthy syntactic patterns many problems can arise. The reason these problems arise is because the patterns in PHRAN are context sensitive. Therefore, a pattern may interact with a new pattern in an unanticipated way causing other patterns that previously worked to fail. This problem can be minimized by using a methodical approach to developing the knowledge base; however, because of the combinatoric explosion of possible interactions this will not guarantee a flawless knowledge base. Fortunately, the general failure mechanism seems to be an inability to match the desired pattern. This usually results in a failed parse rather than an incorrectly formed concept and so SPAN could trap these errors; however, the user of the interface would have to make modifications to the knowledge base that would require knowledge of PHRAN and might not be obvious.

6.2.7.1. Incremental development and regression testing

The suggested method to minimize potential problems is to create a file of test sentences. These sentences should each exercise as many features of the knowledge base in combination as possible. Each test sentence should also test different patterns when possible.

A small test knowledge base containing a minimum number of patterns and words should be created and maintained separately. When a new type of pattern-concept pair (PCP) is needed, it should be developed using the small test knowledge base. When the new PCP appears to be working properly, it can then be added to the main knowledge base. The test sentences used to develop the new PCP should then be run against the main knowledge base. If the system passes this limited test, the full file of test sentences should be run against the extended knowledge base. If the system passes this full test then the test sentences for the new PCP should be added to the file of test sentences.

Open research problems in this area include detecting possible interaction between patterns and generating effective test patterns that cover the interacting cases with a reasonable number of patterns.

6.3. PHRAN's output

The PHRAN-SPAN program can be run in three modes.

1. The default mode invokes SPAN, which describes the concepts it recognizes in English. Any failure to understand the output of PHRAN produces no English output.

2. The trace mode produces a full trace of PHRAN's parsing activity.

3. The output-only mode produces the SRL representation of the meaning and additional internal data structures that PHRAN has at the end of the parsing activity.

The output-only mode will be used in discussing the performance of the prototype system because it is succinct and it can be used to explain failed parses as well as successful parses.

Once the PHRAN-SPAN program has been loaded and initialized for the output-only mode, the user simply types in a sentence from the specification in response to PHRAN's prompt #. The following example shows the user's input and the system's output in the output-only mode.

The user's input:

The cpu sends the code word to the peripheral devices.

System's output:

```
((a_component (object peripheral-device1))
 (device (object peripheral-device1))
 (df_val (object code-word1))
 (ordered_bits (object code-word1))
 (word (object code-word1))
 (a_component (object cpu2))
 (cpu (object cpu2)))

( group (object peripheral-device1)
        (member peripheral-devices))

(uni_dir_vtrans (source (a_component *unspecified*))
                (sink (a_component peripheral-device1))
                (info (df_val code-word1))
                (control (a_component cpu2)))
```

In this example, three sets of information are displayed. The first set of information represents the semantic categories for nouns and noun phrases created by PHRAN while parsing the sentence. The contents of this set are in reverse order, *i.e.* the entries at the beginning of the representation, the top here, were created later than those at the end of the representation. There are seven distinct associations contained in this representation, one for each of the semantic categories associated with the nouns or noun phrases recognized by PHRAN. For example, the representations

```
(a_component (object cpu2))
(cpu (object cpu2))
```

are created when PHRAN parses the noun phrase, *the cpu*. PHRAN uses the information it has about the word cpu to create these representations. The entry in PHRAN's knowledge base for *cpu* is

```
(noun: cpu (cpu a_component)).
```

We see that PHRAN creates a list for each semantic category associated with each noun or noun-phrase. Also note, that PHRAN produces a token for each object referenced. This token is simply the word with a number appended to it, *e.g.*, **cpu2** and **code-word1**. The number indicates the occurrence of that particular word during the present parsing session. We can infer from this that the token **cpu1** must exist and that it was created in an earlier sentence during the current session. The token **code-word1** is the result of PHRAN processing the noun phrase *the code word.*

The output that follows the list of referenced objects and precedes the concept is an optional output and only occurs under special circumstances. It is a list of supplementary information associated with the meaning of a concept. In this example, the supplementary information is associated with the use of the plural form in the phrase *peripheral devices.* Other features of this output will be discussed when they occur in the examples.

The last part of the output is the SRL form of the unidirectional value transfer concept:

```
(uni_dir_vtrans (source (a_component *unspecified*))
                (sink (a_component peripheral-device1))
                (info (df_val code-word1))
                (control (a_component cpu2)))
```

The concept, if one is recognized in the parse, will usually be output last. The only exception to this ordering would be if some unrecognizable input occurred in the sentence after the part of the sentence that produced the concept. In that case, the value **nil** or a string of nils might occur at the end of the output, after the concept.

6.4. Examples

Having described the vocabulary available in the prototype and the system's output, various UVT sentences will be used to demonstrate the system's ability to understand sentences that might occur in specifications.

6.4.1. The Value Transfers

There are three basic value transfer concepts, the UVT, BVT and the NVT. The UVT and the BVT concepts cover the behavior of communicating processes and the NVT is a degenerate case where the focus is on the operation being performed rather than the value being transferred.

6.4.2. The UVT

The basic sentences that describe a UVT concept are

```
<component><UVT-verb><data value>

<component><UVT-verb><component><data value>

<component><UVT-verb><data value>to<component>

<component><UVT-verb><data value>from<component>

<component><UVT-verb><data value>from<component>
to<component>

<component><UVT-verb><data value>on<component>

<component><UVT-verb><data value>in<component>
```

where the <UVT-verb> maybe one of the following:

```
access    check    sample    send    transfer    fetch
```

The <UVT-verb> based patterns described are all active voice patterns; in many cases, the system will also understand the same sentence in the passive voice. Also, additonal adverbial phrases may be added to the basic patterns to indicate timing or sequencing information and structural information.

Examples of English sentences containing <UVT-verbs> that the system understands and the system's output follow. These sentences were all taken from actual specifications and are unaltered.

6.4.2.1. UVT sentence #1

The first example sentence is in the passive voice and contains two interesting phrasal constructions.

A block of data bytes is transferred by a sequence of data cycles.

In response to this sentence the system produces the following output:

```
((df_val (object block-of-data-of-bytes1))
 (unit_grouping_of_bits (object block-of-data-of-bytes1))
 (byte (object block-of-data-of-bytes1)))

(group (object nil) (member byte))
(group (object nil) (member cycle))


(uni_dir_vtrans (source (a_component *unspecified*))
                (sink (a_component *unspecified*))
                (info (df_val block-of-data-of-bytes1))
                (control (a_component *unspecified*))
                (ts_loop sequence-of-data-cycles))
```

The first noun phrase, *a block of data bytes* matches a pattern <structured-object>of<data-object> that describes a structured data object. Unlike the ordinary noun phrase patterns, the concept associated with this pattern contains this additional information. The information is encoded in the token created for the noun phrase by concatenating the word *of* into the string as well as the words describing the individual data objects.

Many of the slots in the UVT concept remain unspecified because this sentence only describes the temporal nature of the data transfer. Additional sentences would have to describe the **source**, **sink** and the **control** which are **a_components** or the existing sentence could be modified to include this information. For example, the phrase *by the secondary storage device* could be added to this sentence to indicate the controlling agent or the phrases *from the peripheral device to the memory unit* to indicate the source and sink of the data.

The final phrase, *by a sequence of data cycles* is an adverbial phrase describing how the transfer is accomplished. A pattern

sequence of <ts_interval>

matches this phrase and modifies the current concept if it is a UVT or BVT concept. The modification is simply to append the timing information to the UVT concept as seen in the systems output. This phrase is very specific and will only match the word *sequence* followed by the word *of* and finally any word which belongs to the semantic category *ts_interval.*

The supplementary information that the words *bytes* and *cycles* referred to more than one object of this type is included in the output in two separate data structures.

No use is currently made of the fact that the indefinite article, *a*, modified *block* and *sequence.* The system makes no inference about the possible intent but simply associates an unknown number of *blocks* with an unknown number of *sequences.* It would be more desirable to use better qualified nouns in describing a system's behavior. For example, the word *every* or *each* could be used in this sentence to improve the intent and remove any ambiguity.

6.4.2.2. UVT sentence #2

The second UVT sentence demonstrates the system's ability to handle a noun phrase that contains a descriptive name embedded in the phrase and a relative clause that also contains a descriptive name.

The peripheral equipment shall sample the !EF code word which is on the !OD lines.

In response to this sentence the system produces the following output:

```
((a_component (object !od-lines1))
 (line (object !od-lines1))
 (df_val (object !ef-code-word1))
 (ordered_bits (object !ef-code-word1))
 (word (object !ef-code-word1))
 (a_component (object peripheral-equipment1))
 (equipment (object peripheral-equipment1)))

(group (object nil) (member line))
(v_c_n_r (df_val !ef-code-word1)
         (a_component !od-lines1)
         (ts_interval *unspecified*))


(uni_dir_vtrans (source (a_component *unspecified*))
                (sink (a_component peripheral-equipment1))
                (info (df_val !ef-code-word1))
                (control (a_component *unspecified*)))
```

One of the problems in natural language processing is how to handle items that are not present in the lexicon. This problem occurs frequently in specifications where acronyms are often created to refer to specific entities. Our solution to this problem was described in Chapter 5, *i.e.*, we simply require the user to preface any lexical item created to aid in describing the system by an exclamation point, !. Otherwise, the system will request the user to re-enter the word and store it as an undefined word. If PHRAN encounters an undefined word, the parse is likely to fail. Other issues associated with this approach and different approaches to the problem are discussed in Chapter 7.

After the system has identified the user defined names *!EF* and *!OD*, it prefixes them to the noun phrase that follows them. In the list of tokens in the system output, we see the tokens **!od-lines1** and **!ef-code-word1**. These tokens can be examined by SPAN and when the exclamation point identifying the user defined name is detected, special processing can be done to group this occurrence with other objects referenced by the same name.

Note, that in processing this sentence the system created a VCNR concept in addition to the main UVT concept. SPAN can use the information in the two concepts to link them together by identifying the token **!ef-code-word1** that occurs in both the VCNR and the UVT.

6.4.2.3. UVT sentence #3

The third example sentence illustrates the system's response to an unqualified noun, *parity*:

The !External register shall check parity on the !EDB during a write.

In response to this sentence the system produces the following output:

```
((ts_interval (object write2))
 (df_opn (object write2))
 (operation-of-writing (object write2))
 (write (object write2))
 (a_component (object !edb-line1))
 (line (object !edb-line1))
 (a_component (object !external-register1))
 (register (object !external-register1)))

(uni_dir_vtrans
     (source (a_component *unspecified*))
     (sink (a_component !external-register1))
     (info)
     (control (a_component !external-register1))
     (v_c_n_r (df_val *unspecified*)
              (a_component !edb-line1)
              (ts_interval *unspecified*))
```

```
(dual_temporal_rel
        (ts_event_init_1 (event_name write2)
                         (event_type *beta*))
        (ts_event_term_1 (event_name write2)
                         (event_type *mu*))
        (ts_arc_1 (arc_type *constraint*)
                  (arc_tail *beta*)
                  (arc_head *pred-init*)
                  (arc_rel gt)
                  (arc_len 0))
        (ts_arc_2 (arc_type *constraint*)
                  (arc_tail *pred-term*)
                  (arc_head *mu*)
                  (arc_rel gt)p
                  (arc_len 0))))
```

Note, that in the third line of the UVT concept, the slot for info is not ***unspecified*** but is unfilled. This is because *parity* was unqualified and therefore, not interpreted as a noun phrase or a **df_val**. This fact can be confirmed by examining the list of tokens at the beginning or top of the output and noticing that no token for parity is found. There are two ways to fix this problem. The first is to fix the the parser to handle such unqualified nouns and the second is to require the user to qualify all nouns with a definite article or another quantifier like all, each or every. Here we modify the sentence by adding the word *the* in front of the word *parity* when re-entering the sentence.

The first part of the new output is shown:

```
((ts_interval (object write3))
 (df_opn (object write3))
 (operation-of-writing (object write3))
 (write (object write3))
 (a_component (object !edb-line2))
 (df_val (object parity2))
 (code_bit (object parity2))
 (parity (object parity2))
 (a_component (object !external-register2))
 (register (object !external-register2)))

(uni_dir_vtrans (source (a_component *unspecified*))
                (sink (a_component !external-register2))
                (info (df_val parity2))
                (control (a_component !external-register1))
                (v_c_n_r (df_val *unspecified*)
                         (a_component !edb-line2)
                         (ts_interval *unspecified*))
                (dual_temporal_rel
                       (ts_event_init_1 (event_name write3)
                                        (event_type *beta*))
          . . .
```

In this example, the auxiliary verb *shall* is processed by PHRAN and does not occur in the final concept; however, the verb *shall* can be used to avoid producing an ambiguous sentence as discussed in Section 5.4.1.

6.5. The Prototype System's Capabilities

One way to characterize the systems capability is to describe the different types of phrases and sentences that it can parse successfully.

Since PHRAN recognizes the basic pattern of

`<noun-phrase><verb>`

any noun phrase followed by a UVT, BVT, NVT, CTI or CTT verb will be parsed successfully by the system. Currently SPAN only understands the UVT and BVT concepts and therefore, cannot produce English output for the other verbs.

The noun phrase may be arbitrarily complex; however, the system currently recognizes only qualified noun phrases, that is a determiner or qualifier must be the first word in the noun phrase.

The remainder of the noun phrase may be a series of from one to five nouns. These nouns maybe modified by an adjective or quantified by a number. The following noun phrases demonstrate some of the types of noun phrases that the system parses successfully:

1. the cpu,
2. the main cpu,
3. the slowest input output device,
4. the two inactive processes, and
5. each peripheral control processor.

Additional examples that contain only nouns can be found in Section 5.4.1.

Most of the UVT, BVT, NVT, CTI and CTT verbs also may have a direct object. As with the subject of the sentence the direct object may be an arbitrarily complex noun phrase like the ones described.

In addition to the basic sentence structure, other phrases and clauses may be added to the basic structure to express additional facts about the behavior being described. The following sentences are examples of the types of sentences that the prototype system can process:

1. *The cpu sends the data to the memory.*
2. *A block of data bytes is transferred by a sequence of data cycles.*

3. *The peripheral equipment shall sample the EF code word which is on the OD lines.*

4. *Each requestor communicates with the arbiter via two lines, a request line and a grant line.*

5. *Select shall be dropped 100 ns after the write is begun.*

6. *The transmitting equipment shall send the word to receiving equipment.*

7. *The cpu computes the difference of !a and !b.*

8. *!UnitA is a cpu.*

9. *The computer shall clear the !ODA line before placing the next word on the !OD lines.*

10. *The cpu starts the memory data transfer activity.*

11. *Upon receipt of the flag, the cpu sends the data to the device.*

12. *The !External register shall check parity on the !EDB during a write.*

Chapter 7

Contributions, Conclusions and Future Work

7.1. Summary of contributions and conclusions

In Chapter 2 on related research, we observed that none of the current techniques for specifying or describing systems were adequate for specifying the behavior of hardware at the system level for the purpose of automated synthesis. We believe that an approach based on mapping a natural language specification of a digital system into a formal model of system behavior is viable and that the prototype system demonstrates the concept.

This research has identified the principal concepts required to specify the behavior of digital systems at a level above the register transfer level, and we believe this to be the major contribution of this research. Such concepts could be used in constructing graphical or formal language interfaces as well as natural language interfaces. Although the model used in our research can be used to describe RTL behavior or even detailed levels of transistor behavior, we believe that current state of the art languages like VHDL and CIRCAL can be used at the lower levels of detail.

In addition to the high level concepts of system behavior, the underlying formal model, the DDS, was refined and the semantics were defined by examples of the various structures. In particular, the basic constructs in the timing and sequencing subspace were extended to allow modelling of asynchronous and synchronous events, causal relationships between events, constraints between events and various combinations of these features.

We also explored the application of natural language processing to aid the specifier in building a formal specification. We developed a prototype system that analyzes single sentences and produces subgraphs in the DDS. We have demonstrated that a knowledge based system for natural language processing like PHRAN is an excellent tool for building a system prototype. Furthermore, using PHRAN allowed us to focus on the more difficult problems of parsing and to develop requirements for a parser in this domain.

Other contributions were the development of a vocabulary for the domain of digital system behavior, a taxonomy of concurent asynchronous behavior, an approach to resolving ambiguity through interaction with the user, and an approch to writing system specifications in English that could be processed by a computer.

7.2. Directions for future research

Two extensions to this work are the expansion of PHRAN's lexicon to encompass more of the vocabulary and a wider range of sentences and the extension of PHRAN-SPAN to process connected text. Though we are reasonably confident that the basic concepts introduced to describe system behavior are probably a complete set, only experimentation with a larger vocabulary and more sentences can validate this claim. With regard to processing connected text, a key research problem is the ability to efficiently *glue* together the subgraphs and fragments of subgraphs produced while processing individual sentences. The use of compiler technology to handle the named objects and the tokens would be the first step. Other work of a similar nature done by Sowa and at IBM on a similar problem are encouraging and suggest that this problem may not be as difficult as originally anticipated. The major concern is the computational complexity of the task, which could be exponential. We believe our modular, block-like approach suggested for writing the specification will render this problem tractable by limiting the global searches for value correspondence.

Another open problem is the habitability issue. Simply stated, this problem arises from the fact that if the system will almost accept normal English as input, the user of the system will have a difficult time recognizing what the system will accept and what it fails to understand. The result is that the user keeps straying over the boundary and the system is to frustrating to use. The only way to determine the habitability is to perform controlled experiments with a large number of users. No experiments have been published on a state of the art natural language system on a limited domain. A restricted input language would reduce this problem.

Also, in the area of natural language processing it would be interesting to use some of the other components of UC with PHRAN--the context model, ellipsis and anophora routines and the text generator could be used to produce a much more sophisticated prototype and to explore some of the other research issues for a natural language specification processor.

Since natural language processing of specifications will probably require significantly more research, a more immediate solution to the specification problem might be to build a *formal* language based on the concepts of system behavior identified in this research.

References

[Abbott 83] Abbott, R.J.
Program Description by Informal English Descriptions.
CACM 26(31):882-894, November, 1983.

[AdvancedMicroDevices 80]
Advanced Micro Devices.
AmZ800 Family Data Book.
Advanced Micro Devices, Sunnyvale, California, 1980.

[Agerwala 79] Agerwala, T.
Putting Petri Nets to Work.
Computer :85-93, December, 1979.

[Allen 83] Allen, J.F.
Maintaining Knowledge about Temporal Intervals.
CACM 26(11):832-843, November, 1983.

[Andler 79] Andler, S.
Predicate Path Expressions.
PhD thesis, Carnegie Melon University, 1979.

[Andrews 83] Andrews, G.R.
Concepts and Notations for Concurrent Programming.
Computing Surveys 15(1):3-43, March, 1983.

[Arens 86] Arens.
CLUSTER: An Approach to Contextual Language Understanding.
PhD thesis, University of California, Berkeley, 1986.

[Arfarmanesh 85]
Afarmanesh, H., D. Knapp, D. McLeod, and A. Parker.
An Extensible Approach to Databases for VLSI CAD.
DISC 85-3, University of Southern California, April, 1985.

[Balzer 85] Balzer, R.
A 15 Year Perspective on Automatic Programming (Invited Paper).
IEEE Transactions on Software Engineering SE-11(11):1257-1268, November. 1985.

[Barbacci 77] Barbacci, M. and D. Siewiorek.
An Architectural Research Facility - ISP Descriptions, Simulation, Data Collection.
In *AFIPS Proceedings 46*, pages 161-173. 1977.

[Barbacci 79a] Barbacci, M.R., G.E. Barnes, R.G. Catell, and D.P. Siewiorek.
The Symbolic Manipulation of Computer Descriptions; The ISPS Computer Description Language.
Technical Report, Carnegie Mellon University, August, 1979.

[Barbacci 79b] Barbacci, M.R., W.B. Dietz and L.J. Szewerenko.
Specifications, Evaluation, and Validation of Computer Architecture Using Instruction Set Processor Descriptions.
In *Proceedings of the 4th International Symposium on Computer Hardware Description Languages*, pages 14-20. IEEE, October, 1979.

[Barbacci 85] Barbacci, M. and T. Uehara (eds.).
Hardware Description Languages.
Computer 18(2), February, 1985.

[Bobrow 75] Boborow, D.G. and A. Collins (eds.).
Representation and Understanding: Studies in Cognitive Science.
Academic Press, Orlando, Florida, 1975.

[Bochmann 82] Bochmann, G.V.
Hardware Specification with Temoral Logic: An Example.
IEEE Transactions on Computers C-31(3):223-231, March, 1982.

[Bolour 82] Bolour, A., T.L. Anderson, L.J. Dekeyse and H.K.T. Wong.
The Role of Time in Information Processing: A Survey.
SIGART Newsletter 80(80):28-48, April, 1982.

[Booth 81] Booth, A.W.
Computer Generated Timing Diagrams to Supplement Simulation .
In Breuer, M., Hartenstein, R. (editors), *Computer Hardware Description Languages and their Applications*, pages 145-152. North-Holland Publishing Company, 1981.

[Breuer 76] Breuer, M.A. and A.D. Friedman.
Diagnosis & Reliable Design of Digital Systems.
Computer Science Press, Inc., Rockville, Maryland, 1976.

[Brinch Hansen 73]
Brinch Hansen, P.
Operating System Principles.
Prentice-Hall, Inc., Englewood Cliffs, N.J., 1973.

[Burstall 81] Burstall, R.M. and J.A. Goguen.
An informal introduction to specifications using Clear.
In Boyer, R.S., Moore, J.S. (editors), *The Correctness Problem in Computer Science*, pages 185-213. Academic Press, New York, N.Y., 1981.

[Cohen 84] Cohen, D.
1984
Private Communication.

[Cohen 86] Cohen, B., W.T. Harwood, M.I. Jackson.
The Specification of Complex Systems.
Addison-Wesley Publishing Company, 1986.

[Comer 79] Comer, J.R.
An Experimental Natural-Language Processor for Generating Data Type Specifications.
PhD thesis, Texas A&M University, May, 1979.
Computing Science.

[Conway 63] Conway, M.E.
A Multiprocessor System Design.
In *AFIPS, Volume 24, Proceedings of the Fall Joint Computer Conference*, pages 139-146. 1963.

[Davis 82] Davis, A.L. and R.M. Keller.
Data Flow Program Graphs.
Computer 15(2):26-41, February, 1982.

[DEC 79] *PDP 11 Bus Handbook*
Digital Equipment Corporation, 1979.

[Dennis 74] Dennis, J.B.
First Version of a Data Flow Procedure Language.
In Robinet, B. (editor), *Lecture Notes in Computer Science.* Volume 19: *Proceedings, Colloque sur la Programmation*, pages 362-376. Springer-Verlag. Berlin, 1974.

[Dewey 84] Dewey, A.
The VHSIC Hardware Description Language.
VLSI Design V(11):33-39, November, 1984.

[Dietmeyer 74] Dietmeyer, D.L. .
Introducing DDL.
Computer :34-38, December, 1974.

[Dijkstra 68] Dijkstra, E.W.
Cooperating sequential processes.
In Genuys, F. (editors), *Programming Languages*, pages 43-112. Academic Press, London, 1968.

[Dijkstra 81] Dijkstra, E.W.
Introduction Why correctness must be a mathematical concern.
In Boyer, R.S., Moore, J.S. (editors), *The Correctness Problem in Computer Science*, pages 1-8. Academic Press, New York, N.Y., 1981.

[Dudani 80] Dudani, S.J.
Generalized Methods for Hardware Description.
PhD thesis, Syracuse University, 1980.

[Duley 68] Duley, J.R., and D.L. Dietmeyer.
A digital System Design Language (DDL).
IEEE Transactions on Computers C-17(9):850-861, September, 1968.

[Dyer 83] Dyer, M.G.
In-Depth Understanding: A Computer Model of Integrated Processing for Narrative Comprehension.
The MIT Press, Cambrige, Massachusetts and London, England, 1983.

[Filman 84] Filman, R.E. and D.P. Friedman.
Coordinated Computing: Tools and Techniques for Distributed Software.
McGraw Hill, New York, N.Y., 1984.

[Findler 79] Findler, N.V. (ed.).
Associative Networks: Representation and Use of Knowledge by Computers.
Academic Press, Orlando, Florida, 1979.

[Fink 85] Fink, P.A., A.H. Sigmon and A.W. Biermann.
Computer Control via Limited Natural Language.
IEEE Transactions on Systems, Man, and Cybernetics SMC-15(1):54-68, January/February, 1985.

[Floyd 67] Floyd, R.W.
Assigning Meaning to Programs.
In Schwartz, J.T. (editor), *Mathematical Aspects of Computing*, pages 19-32. American Mathematical Society, 1967.

[Fujita 85] Fujita, M., H. Tanaka, T. Moto-oka.
Logic Design Assistance with Temporal Logic.
In Koomen, C.J. and T. Moto-oka (editor), *Computer Hardware Description Languages and their applications*, pages 129-138. August, 1985.
Tokyo, Japan.

[Gehani 86] Gehani, M. and A.D. McGettrick (eds.).
International Computer Science Series: Software Specification Techniques.
Addison-Wesely, Workingham, England, 1986.

[Gershman 79] Gershman, A.V.
Knowledge-Based Parsing.
PhD thesis, Yale University, April, 1979.
also available as a Tecnical Report #156.

[Ginsparg 77] Ginsparg, J.M.
A Parser for English and Its Application in an Automatic Programming System.
PhD thesis, Stanford University, June, 1977.

[Goguen 77] Goguen, J.A., J.W. Thatcher, E.G. Wagner and J.B. Wright.
Initial Algebra Semantics and Continuous Algebras.
Journal of the Association for Computing Machinery 24(1):68-95, January, 1977.

[Goguen 79] Goguen, J.A. and J.J. Tardo.
An Introduction to OBJ: A Language for Writing and Testing Algebraic Program Specifications.
In *Procedings of the Specification of Reliable Software Conference*, pages 170-189. IEEE Computer Society, 1979.

[Goldman 82] Goldman, N.M. and D.S. Wile.
GIST LANGUAGE DESCRIPTION
Information Sciences Institute, 1982.

[Gordon 81a] Gordon, M.
A Very Simple Model of Sequential Behavior of nMos.
In *VLSI 81*, pages 85-94. 1981.

[Gordon 81b] Gordon, M.J.C.
Register Transfers and Their Behaviors.
In *Computer Hardware Description Languages and Their Applications*, pages 23-36. IFIP, 1981.

[Gostelow 77] Gostelow, K.P.
Flow of Control, Resource Allocation and the Proper Termination of Programs.
PhD thesis, University of California, Los Angeles, December, 1977.

[Granacki 85] Granacki, J., D. Knapp and A. Parker.
The ADAM Design Automation System: Overview, Planner and Natural Language Interface.
In *Proceedings of the 22nd ACM/IEEE Design Automation Conference*, pages 727-730. ACM/IEEE, June, 1985.

[Grinberg 80] Grinberg, M.R.
A Knowledge-Based Design Environment for Digital Electronics.
PhD thesis, University of Maryland, 1980.
Dept. of Computer Science.

[Hafer 81]
Hafer, L.
Automated Data-Memory Synthesis: A Formal Model for the Specification, Analysis and Design of Register-Transfer Level Digital Logic.
PhD thesis, Carnegie Mellon University, May, 1981.
Dept. of Electrical Engineering.

[Hailpern 80]
Hailpern, B.T. and S.S. Owicki.
Verifying Network Protocols Using Temporal Logic.
Technical Report 192, Stanford University, June, 1980.

[Hayes 78]
Hayes, J.P.
Computer Architecture and Organization.
McGraw-Hill Book Company, New York, N.Y., 1978.

[Hayes 83]
Hayes, P.J. and J.G. Carbonell.
A Tutorial on Techniques for Natural Language Processing.
CMU-CS 83-158, Carnegie Mellon University, October, 1983.

[Heninger 80]
Heninger, K.L.
Specifying Software Requirements for Complex Systems: New Techniques and Their Applications.
IEEE Transactions on Software Engineering SE-6(1):2-13, January, 1980.

[Hill 79]
Hill, D.D.
ADLIB: A Modular, Strongly-Typed Computer Design Language.
In *Proceedings 4th International Symposium on Computer Hardware Description Languages*, pages 75-81. IEEE, October, 1979.

[Hoare 78]
Hoare, C.A.R.
Communicating Sequential Processes.
Communications of the ACM 21(8):666-677, August, 1978.

[Huang 81]
Huang, C.
Computer-Aided Logic Synthesis Based on a New Multi-Level Hardware Design Language—LALSD II.
PhD thesis, School of Advanced Technology, State University of New York At Binghahamton, 1981.

[IBM 74] IBM Electronic Systems Center.
Storage Controller Interface.
1974
Prelimnary Specification, 16 January 1974.

[IEEE 84]
IEEE Guide to Software Requirements Specifications.
Published by the IEEE.
1984
ANSI/IEEE Std 830-1984, February 10, 1984.

[Intel 84] Intel.
MULTIBUS II Bus Architecture Specification Handbook.
Intel Corporation, Santa Clara, CA.
1984

[Intermetrics 85] *VHDL User's Manual, Volume 1 - Tutorial*
Intermetrics Inc., 1985.
Intermetrics Report IR-MD-065-1.

[Jensen 85] Jensen, K. and N. Wirth.
Pascal: User Manual and Report Third Edition.
Springer-Verlag, New York Berlin Heidelberg Tokyo, 1985.

[Kahn 79] Kahn, Giles (editor).
Semantics of Concurrent Computation, Proceedings of the International Symposium, Evian, France, July 2-4, 1979.
Springer-Verlag, Berlin Heidelberg New York, 1979.
Lecture Notes in Computer Science 70.

[Knapp 83] Knapp, D.W. and A.C. Parker.
A Data Structure for VLSI Synthesis and Verification.
DISC 83-6, University of Southern California, October, 1983.

[Knapp 84] Knapp, D.W.
AGIS - User Manual
1984.
Unpublished.

[Knapp 85] Knapp, D.W. and A.C. Parker.
A Unified Representation for Design Information.
In Koomen, C.J. and T. Moto-oka (editor), *Computer Hardware Description Languages and their applications*, pages 337-353. IFIP, August, 1985.
Tokyo, Japan.

[Koomen 85] Koomen, C.J. and T. Moto-oka (editor).
IFIP Seventh International Conference on Computer Hardware Description Languages and their Applications, Tokyo, Japan, 29-31 August, 1985.
North-Holland, Amsterdam New York Oxford, 1985.

[Kumar 82] Kumar, K.S.
DTMS II -- An Improved Computer Hardware Description Language for Modules and Systems.
PhD thesis, Kansas State University, 1982.
Dept. of Electrical Engineering.

[Lamport 83] Lamport, L.
What Good Is Temporal Logic.
In Mason, R.E.A. (editor), *Information Processing 83*, pages 657-658. 1983.

[Lipovski 78] Lipovski, G.J.and K.L. Doty.
Developments and Directions in Computer Architecture.
Computer :54-67, August, 1978.

[Liskov 79] Liskov, B.H. and V. Berzina.
An Appraisal of Program Specifications.
In Wegner, P. (editor), *Research Directions in Software Technology*, pages 277-301. MIT Press, Cambridge, Massachusetts and London, England, 1979.

[London 82] London, P.E. and M.S. Feather.
Implementing Specification Freedoms.
In *Science of Computer Programming 2*, pages 91-131. North-Holland Publihing Company, 1982.

[Mander 79] Mander, K.C. and S.G. Presland.
An Introduction to Specification Analysis -SPAN.
Technical Report CSS/79/12, University of Liverpool (Gt. Brit), 1979.
Departmentof Computational and Statistical Science.

[Marcus 80] Marcus, M.P.
A Theory of Syntactic Recognition for Natural Language.
The MIT Press, Cambridge, Massachusetts and London, England, 1980.
Based on the author's thesis 1979.

[Matty 83] Matty, D.G.
Constraint Driven Synthesis of Hardware Design.
PhD thesis, University of Utah, 1983.
Dept. of Computer Science.

[McCrimmon 63] McCrimmon, J.M.
Writing with a Purpose.
Houghton Mifflin Company, Boston, 1963.

[McFarland 81] McFarland, M.C.
Mathematical Models for Verification in a Design Automation System.
PhD thesis, Carnegie Mellon University, 1981.
Dept. of Electrical Engineering.

[Milne 83] Milne, G.J.
CIRCAL: A calculus for circuit description.
INTEGRATION, The VLSI Journal 1(2 & 3):121-160, 1983.

[Milner 80] Milner, R.
A Calculus of Communicating Systems.
Springer-Verlag, Berlin Heidelberg New York, 1980.
Lecture Notes in Computer Science 92.

[Milner 83] Milner, R.
Calculi for Synchrony and Asynchrony.
In *Theoretical Computer Science 25*, pages 267-310. North-Holland, 1983.

[Minsky 68] Minsky, M. (ed.).
Semantic Information Processing.
The MIT Press, Cambridge, Massachusetts and London, England, 1968.

[Moore 82] Moore, V.S.
A High-Level Language Based VLSI Design System.
PhD thesis, The University of Florida, 1982.
Dept. of Electrical Engineering.

[Moszkowski 83] Moszkowski, B.
A Temporal Logic for Multilevel Reasoning about Hardware.
In Uehara, T., Barbacci, M. (editor), *Computer Hardware Description Languages and their Applications*, pages 79-90. North-Holland Publishing Company, 1983.

[Nagle 81] Nagle, A.W.
Automated Design of Digital-System Control Sequencers from Register-Transfer Specifications.
PhD thesis, Carnegie Mellon University, 1981.

[Nash 86] Nash, J.D. and L.F. Saunders.
VHDL Critique.
IEEE Design & Test of Computers 3(2):54-65, April, 1986.

[Paker 83] Paker, Y. and J.P. Verjus (eds.).
Distributed Computing Systems: Synchronization, Control and Communication.
Academic Press, Orlando, Florida, 1983.

[Parker 79a] Parker, A.C. et al.
The CMU Design Automation System.
In *Proceedings 16th Design Automation Conference*, pages 73-80. ACM/IEEE, June, 1979.

[Parker 79b] Parker, A.C., D.E. Thomas, S. Crocker, and R.G.G. Catell.
ISPS: A Reteospective View.
In *Proceedings 4th International Symposium on Computer Hardware Description Languages*, pages 21-27. IEEE, October, 1979.

[Parker 81] Parker, A.C. and J.J. Wallace.
An I/O Hardware Description Language.
IEEE Transactions on Computers C-30(6):423-428, June, 1981.

[Parker 83] Parker, A.C.
Simulation Effectiveness Research Report.
DISC Report 83-2, University of Southern California, April, 1983.
Dept. of Electrical Engineering-Systems.

[Parker 84] Parker, A.C. and N.B. Park.
SLIDE+ Language Reference Manual
1984.
Unpublished.

[Pavlovic-Lazetic 86]
Pavlovic-Lazetic, G. and E. Wong.
Managing Text as Data.
In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 111-116. 1986.

[Perrin 59] Perrin, P.G.
Writer's Guide and Index to English.
Scott, Foresman and Company, 1959.

[Peterson 77] Peterson, J.L.
Petri Nets.
Computing Surveys 9(3):223-252, September, 1977.

[Pnueli 77] Pnueli, A.
The Temporal Logic of Programs.
In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46-57. October, 1977.

[Rescher 71] Rescher, N. and A. Urquart.
Temporal Logic.
Springer-Verlag, New York - Wein, 1971.

[Saunders 85] Saunders, L.F.
VHDL Examples for the IEEE Working Group on System Design of thbe Design Automation Standards Subcommittee of the DATC.
1985
October 15, 1985.

[Schank 73] Schank, R.C. and K.M. Colby (eds.).
Computer Models of Thought and Language.
W.H. Freeman and Company, SanFrancisco, CA, 1973.

[Schank 75] Schank, R.C.
Fundamental Studies in Computer Science. Volume 3: *Conceptual Information Processing.*
American Elsevier Publishing Co., New York, N.Y., 1975.

[Schank 77] Schank, R.C. and R.P. Abelson.
Scripts, Plans, Goals and Understanding: An Inquiry into Human Knowledge Structures.
Lawerence Erlbaum Associates, Publishers, Hillsdale, New Jersey, 1977.

[Schank 81] Schank, R.C. and C.K. Riesbeck.
Inside Computer Understanding: Five Programs Plus Minatures.
Lawerence erlbaum Associates, Publishers, Hillsdale, New Jersey, 1981.

[Schwartz 83] Schwartz, R.L., P.M. Melliar-Smith and Fredrich Vogt.
An Interval-Based Temporal Logic.
In *Logic of Programs, Workshop*, pages 443-457. 1983.
Lecture Notes in Computer Science 164.

[Shahdad 85] Shahdad, M. et.al.
VHSIC Hardware Description Language.
Computer 18(2):94-103, February, 1985.

[Shapiro 83] Shapiro, V.
On Formal Verification of Systems Described by A Graph Model of Behavior.
Master's thesis, University of California, Los Angeles, June, 1983.
Computer Science Department.

[Shaw 80] Shaw, A.C.
Software Specification Languages Based on Regular Expressions.
In Riddle, W., Fairley, R. (editors), *Software Development Tools*, pages 148-175. Springer-Verlag, New York, N.Y., 1980.

[Shiva 79] Shiva, G.S.
Computer Hardware Description Languages--A Tutorial.
Proceedings of the IEEE 67(12):1605-1615, December, 1979.

[Singh 81] Singh, G.S.
Description techniques for Digital Systems Containing Complex Components.
PhD thesis, Kansas State University, 1981.
Dept. of Electrical Engineering.

[Sorensen 78] Sorensen, I.H.
System Modelling.
Master's thesis, University of Aarhus, Denmark, March, 1978.
Computer Science Department.

[Sowa 84] Sowa, J.F.
The System Programming Series: Conceptual Structures: Information Processing in Mind and Machine.
Addison-Wesley Publishing Company, 1984.

[Sowa 86] Sowa, J.F. and E.C. Way.
Implementing a Semantic Interpreter Using Conceptual Graphs.
IBM Journal Research and Development 30(1):57-69, January, 1986.

[Strom 83] Strom, R.E. and S. Yemini.
NIL: An Integrated Language and System for Distributed Programming.
In *Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, pages 73-82. ACM, June, 1983.

[Suzuki 84] Suzuki, N.
Experience with Specification and Verification Using PROLOG.
In Kunii (editor), *Lecture Notes in Computer Science.* Volume 163: *VLSI Engineering: Beyond Software Engineering*. Springer-Verlag. Tokyo, Japan, 1984.

[Tennant 81] Tennant, H.
Natural Language Processing.
Petrocelli Books, Inc., New York and Princeton, 1981.

[Tesler 68] Tesler, L.G. and H.J. Enea.
A Language Design for Concurrent Processes.
In *AFIPS, Proceedings of the Spring Joint Computer Conference*, pages 403-408. 1968.

[Tiechroew 77] Tiechroew, D. and E.A. Hershey III.
PSL/PSA: A Computer-Aided Documentation and Analysis of Information Processing.
IEEE Transactions on Software Engineering SE-3(1):41-48, January, 1977.

[Uehara 81] Uehara, T. et al.
DDL Verifier and Temporal Logic.
In Breuer, M. and R. Hartenstein (editors), *Computer Description Languages and their Applications*, pages 51-61. IFIP, 1981.

[Uehara 83] Uehara, T., Barbacci, M. (editor).
IFIP Sixth International Symposium on Computer Hardware Description Languages and their Applications, 1983.
North-Holland Publishing Co., Amsterdam, New York, Oxford, 1983.

[USN 73] Department of the Navy, Naval Ship System Command.
Input/Output Interfaces, Standard Digital Data, Navy Systems.
MIL-STD-1397(SHIPS), 30 August 1973.
1973

[van-Mierop 78] van-Mierop, D., S. Crocker and L. Marcus.
Verification of the FTSC Microprogram.
In *Proceedings of the 11th Annual Microprogramming Workshop*. 1978.

[vanCleemput 77]
van Cleemput, W.M.
An Hierarchical Language for the Structural Description of Digital Systems.
In *Proceedings of the 14th Design Automation Conference*, pages 377-385. ACM/IEEE, June, 1977.

[Weischedel 83] Weischedel, R.M.
Prerequisites for Deriving Formal Specifications from Natural Language Requirements: Final Reort.
AFOSR-TR 83-0678, University of Delaware, 1983.
Department of Computer & Information Science.

[Wilensky 80] Wilensky, R. and Y. Arens.
PHRAN -- a Knowledge Based Approach to Natural Language Analysis.
Electronic Research Memorandum UCB M80/34, University of California Berkeley, 1980.

[Wilensky 84] Wilensky, R., Y. Arens and D. Chin.
Talking to UNIX in English: An Overview of UC.
CACM 27(6):574-593, June, 1984.

[Winchester 80] Winchester, Jr., J.W.
Requirements Definition and Its Interface to the SARA Design Methodology for Computer-Based Systems.
PhD thesis, University of California, Los Angeles, December, 1980.
Computer Science Department.

[Winograd 72] Winograd, T.
Understanding Natural Language.
Academic Press, 1972.

[Winston 84] Winston, P.H. and B.K.P. Horn.
LISP.
Addison-Wesley, 1984.

Appendix A

Vocabulary

A/D
ASCII
BCD
BILBO
Booth
CAM
CCD
CDC
CDL
CMOS
CORDIC
CPU
CSA
Cray
D/A
DEC
DMA
Data General
EBCDIC
EPROM
FIFO
FIR
GHz
Gray code
HDL
HOL
Hamming
Hewlett-Packard
Huffman
Hz
I/O
IAS
IBM
IC
IIR
IMP
IO
Illiac IV
Intel
JK
Johnson counter
K
KHz
Kalman
LIFO
LRU
LSI
LSSD
M
MHz
MIMD
MISD
MSI
Motorola
NMOS
PLA
PROM
RALU
RAM
RISC
ROM
SECDED
SIMD
SISD
SLA
TI
TI ASC
Turing machine

UART
VAX
VLSI
Von Neumann
Winchester
abend
abort
absolute address
absolute value
accelerate
accelerator
accept
access
accessory
accompany
accomplish
account
accumulate
accumulator
accuracy
acknowledge
acknowledgement
action
activate
active
activity
acyclic
adapt
adapter
add
adder
additional
address
addressing mode
after
algorithm
alphabetic
alphanumeric
alias
align
alignment
alive
allocate
allocation
allow
alter
alterable
alternate
amass
amplitude
analog
analysis
analyze
and
angle
append
arbiter
arbitrate
arbitration
architecture
arccosine
arcsine
arctangent
arithmetic
arithmetic expression
arithmetic instruction
array
arrival
arrive
ascending
assemble
assembler
assert
associate
associative memory
asynchronous
at
attach
attribute
auxiliary
available
availability
avoid
background

backup
balance
band
bandwidth
bank
base
batch
battery
baud
before
begin
beginning
bias
bidirectional
bifurcate
binary
bipolar
bistable
bit
bit density
bit sliced
block
block transfer
board
boolean
borrow
box
bpi
branch
bubble memory
bucket
buffer
bug
burst
bus
busses
byte
cache
calculate
call
cancel
capability
capable
capacity
card
card cage
carrier
carry
carry save
cell
center
chain
channel
character
characteristic
characterize
charge
check
checkpoint
checker
chip
circuit
circular
classify
clobber
clock
close
cluster
code
collect
collection
collision
combination
combinational
combinatorial
combine
command
commit
commitment
communicate
communication
commutate
commute
compact

compaction
comparator
compare
comparison
compatibility
complement
complex
complexity
component
compose
composition
compress
compression
comprise
computation
compute
concatenate
concurrent
condition
conditional
configuration
configure
conflict
connect
connection
connector
consecutive
consequential
consist
console
constant
constraint
consume
contain
contend
content addressable memory
contention
contiguous
control
controller
conversion
convert
converter
cooperate
coordinate
copy
core
correct
cosine
cost
count
counter
couple
critical
critical path
cross-bar
cross-point
current
current loop
cycle
daisy-chain
data
data flow
data link
data path
deactivate
dead
deadlock
deallocate
decay
decide
decipher
decimal
decision
decode
decoder
decompose
decompress
decrease
decrement
decrypt
dedicate
default
defect

define
delay
delete
deliver
demodulate
demodulator
demultiplex
demultiplexer
demux
depress
derive
describe
designate
destroy
destruction
detect
determine
device
diagnose
diagnostic
die
difference
digit
direct
direct access
direct address
direct mapping cache
direct mapping memory
direct memory access
directory
disassemble
discard
discrete
discriminate
disk
dispatch
dispense
displace
displacement
display
dispose
distinguish
distribute
divide
dividend
divider
divisor
double
double-precision
doble word
down
download
drain
draw
drive
dual
due
duplex
duplicate
during
dyadic
dynamic
eavesdrop
eavesdropper
echo
effect
effective
effuse
elapse
elect
element
eliminate
emit
empty
emulate
emulator
enable
encapsulate
encipher
encode
encoder
encounter
encrypt
end

ending
energize
ensure
enter
entire
entrance
entry
enumerate
enumerative
environment
equal
equalize
equate
equivalent
eradicate
erasable
erase
erasing
error
error-correcting
error-detecting
escape
establish
establishment
estimate
even-numbered
even parity
event
evoke
examination
examine
exceed
except
exception
excess-three
excessive
exchange
exclude
exclusion
exclusive
exclusive nor
exclusive or
execute
execution
execution time
executive
exemplify
exempt
exemption
exhaust
exhibit
exist
existence
existent
exit
exit point
expand
expansion
expect
expectable
expectation
expedite
expire
explain
explanation
explicate
explicative
explicit
exploit
exponent
exponentiate
export
express
expression
expunge
extend
extension
external
extract
extraction
extrapolate
extrapolation
extricate
extrude

extrusion
exude
facilitate
factor
fade
fail
fail-safe
fail-soft
failure
failure rate
fall
fall off
fan
fan in
fan out
fashion
fasten
fatal
fault
fault coverage
feed
feedback
fetch
field
file
fill
filter
find
find out
finish
finite state machine
fire
firm
firmware
first
first in first out
first-fit
fix
fixed point
flag
flip
flip flop
float
floating point
flood
flow
flowchart
flush
follow
following
forbid
forbidden list
force
forget
fork
form
format
formulate
forward
fraction
fracture
fragment
frame
free
frequency
full adder
full duplex
full word
function
functional units
furnish
future growth
gain
gap
garble
gate
gather
generate
generator
get
give
global
glom
go

grab
grant
graphics
group
grouping
guard
guardian
guide
half
half duplex
half word
halt
handle
handler
happen
hard disk
hard error
hardware
hardwire
hash
hazard
header
heap
hesitate
hex
hexadecimal
hide
hierarchical
hierarchy
high
high level
high level language
high order
high order bit
histogram
hit
hit ratio
hold
horizontal
housekeeping
identical
identifier
identify
idle
ignore
immediate
immediate addressing
impede
impend
implement
implementation
import
improve
inactivate
inactivity
include
inclusive or
increase
increment
incremental
independent
index
indicate
indicator
indication
indicative
indirect
individually
induce
inequality
infer
information
inherit
inhibit
initialization
initialize
initiate
initiation
input
input/output
insert
insertion
inspect
inspection

instantiate
instantiation
institute
institution
instruct
instruction
instruction set processor
instrument
integer
integral
integrate
integrated circuit
intend
intention
interchange
interchangeable
intercommunication
interface
interfere
interference
interlace
interleave
interlock
intermediate
intermix
internal
interpolate
interpret
interpretation
interrogate
interrupt
interrupt-based
interrupt vector
interval
interval timer
introduce
introduction
inundate
invalid
invalidate
invert
inverter
invoke
invocation
involve
isolate
isolation
item
itemize
iterate
iteration
jam
join
joy stick
jump
justification
justify
keep
key
keyboard
kill
kludge
label
lack
lag
latch
latency
lay
lead
leading
learn
least significant
leave
left
leftjustify
left-to-right
leftmost
length
lengthen
lessen
let
level
liberate
library

life
lift
light
like
limit
line
line noise
line printer
linear
linearize
lines per minute
lines per second
link
linked list
linker
list
listen
listing
literal
live
load
local
locate
location
locator
lock
log
logic
logical
look
look ahead
lookup
loop
low
low order
low order bit
lower
lowercase
lowest
machine
macro
macro
magnetic tape
magnitude
mail
main
main memory
mainframe
maintain
maintenance
major
make
manage
management
manipulate
manipulation
mantissa
map
mapping
mark
mask
mass
master
master-slave
match
matrix
maximize
maximum
may
measure
measurement
mechanize
meet
meeting
memory
memory address register
menu
merge
merger
message
meter
microcode
microprocessor
microprogram

microstep
middle
mini
minimize
minimum
minor
minuend
misaddress
miss
missend
misset
mix
mnemonic
model
moderate
modify
modulate
modulator
monadic
monitor
most significant
mount
move
multiple
multiplex
multiplexer
multiplicand
multiplier
multiply
multiprogram
munch
munge
mutual exclusion
mux
nand
narrow
natural
natural logarithm
natural number
necessitate
need
negate
negative
negotiate
nest
net
network
nibble
niche
nitpick
node
nonabrupt
nonconcurrent
nondestructive
nonlinear
nonrecurring
nonrestoring divide
nonrestoring division
nop
nor
normalize
not
notation
note
notice
null
nullify
number
numerate
numeric
numerical
oblige
observability
observe
obsolete
obstruct
occlude
occupy
occur
octal
odd
odd parity
odd-numbered
off-load

off-loading
off-the-shelf
offend
offer
omit
one-way
ones-complement
online
open
opening
operand
operate
operation
operator
optimize
optimum
or
order
organize
orient
orientation
origin
originate
orphan
oscillate
outflow
outline
output
outweigh
overcommit
overcomplicate
overcurrent
overflow
overlap
overlay
overlook
override
oversee
overshoot
overspend
overvoltage
owe

pace
pack
package
packet
packetize
page
page table
paginate
paint
pair
parallel
parameter
parameterize
parcel
parity
parity bit
parity check
parity error
parse
part
participant
participate
partition
pass
passive
patch
pause
peak
peek
peephole
peg
perform
performance
period
peripheral
permit
permute
perpetual
perpetuate
persist
pertain
perturb

perturbation
petition
phase
pick
picture
piece
pile
pin
pipe
pipeline
pitch
pivot
place
placement
plan
playback
plot
plug
plumb
point
point out
pointer
poke
police
policy
poll
polling
pop
porch
port
portion
position
positional
possess
post
postbox
postfix notation
power
power down
power up
practical
precede
precharge
precision
preclude
precondition
predestine
predetermine
predicate
predispose
preempt
preemption
prefabricate
preface
prefer
preference
prefetch
prefix
prefix notation
preliminary
prename
prep
prepare
preprocess
preprocessor
prerequisite
prescribe
preselect
present
preserve
preset
prestore
pretest
prevent
prevention
previous
primarily
primary
prime
print
print out
printed circuit
prior
priority

private
privelege
priveleged
probable
probe
procedure
proceed
process
processor
produce
product
program
program counter
programmable
programmable logic array
programmable read only memory
progress
progression
project
prompt
proof
proportion
proposal
propose
protect
protection
protocol
protract
prove
provide
provides for
provoke
prune
public
pull
pulse
pump
punctuate
punctuation
purchase
purge
purpose
push
push button
put
pyramid
quadruple
qualify
quality
quantity
quantize
quench
question
queue
quicken
quit
quote
quotient
race
radiate
radix
raise
ramp
random
random access
random access memory
randomize
range
rank
raster
rate
reach
react
reaction
reactivate
read cycle
read only memory
read read reading
read write cycle
read write memory
read-write
readable
reader
readout

ready
real
real-time
reallocate
reason
recalculate
recall
recast
receipt
receive
recharge
recipe
recirculate
recode
recognize
recommend
recommendation
reconfigurable
reconfiguration
reconfigure
reconnect
reconvert
record
recover
recoverable
recovery
recreate
rectifier
rectify
recur
recurrence
redefine
redefinition
redirect
redirection
redistribute
redistribution
redo
reduce
reduction
redundant
rentrant
refer
reference
referent
referential
refine
refinement
reflect
reflection
reform
refresh
refuse
register
regression
regular
regularity
regularize
regulate
regulation
rehash
reject
relate
relational
relative
relax
release
reliability
reliable
relinquish
relocatable
relocate
remain
remainder
remark
remote
remount
removal
remove
renew
reorder
repair
repeat
repeater

replace
replacement
replenish
replicate
replication
reply
report
represent
representation
request
requestor
request-to-send
require
requirement
rescind
resend
reservation table
reserve
reset
reside
resident
resident
resolution
resolve
resort
resource
respective
respond
response
rest
restart
restore
restrain
restraint
restrict
restriction
restructure
result
resume
resumption
resupply
retain
retard
retardation
retest
retrace
retract
retraction
retranslate
retranslation
retrial
retrieval
retrieved
retry
return
reuse
reverse
reverse Polish
revert
review
revise
revision
revocation
revoke
rewake
rewind
rewrite
ribbon
ribbon cable
ride
rig
right
right-to-left
rightmost
ring
ring bus
ring counter
ring network
ripple
rise
rival
roam
robust
roll

roll back
roll over
rollback
round
round-robin
route
routine
rove
rover
row
run
run away
run down
run through
runaway
rundown
runthrough
sample
satisfaction
satisfy
saturate
saturation
save
say
scalar
scale
scale up
scan
scatter
schedule
schematic
scheme
score
scratch
scratch pad
scratchpad memory
screen
scribe
scroll
scrounge
scrub
scrunch
scrutinize
scrutiny
seal
search
secondarily
secondary
section
sector
segment
seize
select
selectable
selection
selector
self-starting
send
sense
sensitize
separable
separate
separation
sequence
sequencer
sequential
serial
serialize
serializer
serve
server
service
set
set associative cache
set up
settle
setup
sever
severable
shadow
shadow cache
shake
shape
shall

shaper
share
shed
shell
shelter
shield
shift
shift register
shift-and-subtract
shift-and-subtract
ship
short
shorten
shorting bar
shove
show
shrink
shrink tubing
shuffle
shunt
shut
shut down
shut off
shutdown
shutoff
shuttle
side
sieve
sift
sign
sign off
sign on
sign-magnitude
signal
significant
silence
silent
simple
simplify
simulate
simulation
simulator
simultaneous
sine
single
single step
single-line
single-line interrupt
sink
siphon
size
skew
skim
skip
slack
slackening
slave
sleep
slew
slew rate
slice
slip
slop
slot
smash
smear
smooth
snatch
snuff
software
solder
solution
solve
sort
source
space
span
spare
speak
speaker
spec
special
special purpose
specification

specify
speed up
speedup
spend
spike
spindle
splice
split
spool
spooler
spot
spurious
square
square root
stability
stabilize
stable
stack
stage
stamp
stand by
stand for
standard
standardize
star
start
starvation
starve
stash
state
state vector
static
status
status register
steal
steer
step
stimulate
stimulation
stitch
stock
stop
storage
store
straddle
straight through
straighten
strain
strap
stream
streamer
stress
stretch
string
strip
strobe
structure
stub
style
subject
submit
submittal
substitute
subsystem
subtask
subtract
subtracter
subtrahend
sum
summer
summon
super computer
super micro
super mini
superimpose
superimposition
superpose
superposition
supersede
supersession
supervise
supervisor
supervisory
supplant

supplement
supply
support
suppress
suppression
surge
suspend
sustain
sustenance
swap
swap in
swap out
swing
switch
symbolic
symmetric
symmetry
synchronize
synchronous
syndrome
synthesis
synthesize
system
systematic
systematical
systematize
systolic
table
tabulate
tag
tail
take
talk
tally
tangent
tap
tape
target
targetable
task
technique
tee
temporary
terminal
terminate
test
testability
testable
text
thin
think
thought
thrash
thread
through
tick
tickle
tie
tighten
time
time-multiplex
time-multiplexed
time-tag
time-tagged
timing
toggle
total
touch
tour
trace
track
trade
trade off
trade-off
trail
trailer
transact
transaction
transceiver
transduce
transducer
transfer
transform
transient

transisit
transistor
transistorize
transit
translate
translation
transmit
transmittance
transmutation
transmute
transparent
transport
transportation
transpose
transposition
transverse
trap
trap-door
trash
travel
traversal
traverse
treat
treatment
tree
trend
trial
trial and error
trick
trickery
trickle
tricky
trifurcate
trigger
trim
trip
triple
triplicate
tristatable
tristate
true
truncate
trunk
trunk line
truth
truth set
truth table
truth-value
try
tunable
tunableness
tune
tune out
tuned-in
tunnel
tuple
turbocharge
turkey
turn
turn off
turn on
turnkey
turtle
turtle graphics
tweak
twiddle
twist
twisted pair
two-out-of-five
two-way
twos-complement
twos-complement
type
typical
typify
typo
typographic
ugly
unable
unacceptable
unbalanced
uncertain
uncertainty
unconditional

undefined
under
underbudgeted
underneath
unequal
uneven
uniform
unilateral
union
uniprocessor
unique
unison
unit
unite
unitialized
units digit
units place
universal
unknown
unlock
unpack
unthread
up
up-and-down
up-to-date
up-to-the-minute
upcoming
update
upgrade
upon
upper
upset
uptime
usable
use
user
user intervention
user oriented
utility
utilize
vacancy
vacant
vacate
valid
validate
validation
value
valve
vanish
vaporize
variability
variable
variance
variant
variation
variety
vary
vector
vector product
vector space
vector sum
vectorize
vendor
vendor supplied
verbose
verboten
verification
verify
vernier
version
vertical
veto
via
viable
victim
vie
virtual
virtual memory
visibility
visible
visit
vocabulary
void
volatile

volatileness
volatility
volt
voltage
vote
voter
vulnerability
vulnerable
wag
wait
wake
waken
wakeup
walk
wander
want
warm
warm up
warn
warning
waste
watch
wave
waveform
wavefront
waveguide
wavelength
way
weaken
weakness
wear out
weave
wedge
weigh
weight
well formed
well-conditioned
when
while
whole
wide
widen
width
wiggle
wind
window
wipe
wire
withdraw
withhold
word
work
workload
wraparound
wring
writable
write
write cycle
write through
wrong
x-axis
x-coordinate
x-intercept
xnor
xor
y-axis
y-coordinate
y-intercept
yank
yield
yo-yo
z-axis
z-coordinate
z-intercept
zero
zero-address
zeroth
zone
zonk
zoom

Appendix B

Examples from Prototype System's PHRAN Database

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;This a test file for developing new PCPs for use with PHRAN/SPAN
;
;John J. Granacki
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(setq *pairs* '(*top* nil))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;NOUNS

(noun: access (access use-of mentob))
(noun: acquisition (acquisition mentob))
(noun: activity activities (activity df_opn))
(noun: address addresses (address df_val a_value))
(noun: agent (agent df_opn a_component))
(noun: alu (alu a_component))
(noun: arbiter (arbiter a_component))
(noun: arbitration (arbitration mentob))
(noun: arbitrator (arbitrator a_component))
(noun: bit (bit binary_digit df_val a_value))
(noun: block (block entity_logical group_of_bits struct_obj
              df_val a_value))
(noun: byte (byte unit_grouping_of_bits df_val a_value))
(noun: bus (logical_group_of_lines a_component))
(noun: cache (cache high-speed-memory a_component))
(noun: call (call request df_opn))
(noun: channel (channel line a_component))
(noun: check (check process df_opn))
(noun: clock (clock ts_generator a_component))
(noun: code (code system_symbols))
(noun: command (command causal_data df_val a_value mentob))
(noun: condition (condition logically-testable-state df_val
                  a_value))
(noun: control (control diecting-process df_opn))
(noun: cosine (cosine result df_val unary_fnc df_opn))
```

```
(noun: cotangent (cotangent result df_val unary_fnc df_opn))
(noun: count (count result df_val binary_fnc df_opn))
(noun: cpu (cpu a_component))
(noun: cycle (cycle time_interval ts_interval))
(noun: data nil (data information df_val a_value))
(noun: delay (delay ts_interval a_deferring mentob))
(noun: device (device a_component))
(noun: difference (difference result df_val binary_fnc df_opn))
(noun: end (end the_end ts_event))
(noun: equipment nil (equipment a_component))
(noun: event (event ts_event))
(noun: fetch (fetch process df_opn))
(noun: fft (fft result df_val binary_fnc df_opn))
(noun: filter (filter selective-process a_component))
(noun: flag (flag indicator-variable df_val a_value))
(noun: format (format defined-arrangement mentob))
(noun: function (function operation df_opn))
(noun: grant (grant permission mentob))
(noun: handshake (handshake protocol procedure))
(noun: index indicies (index  table-of-locations subscript
                      mentob))
(noun: information (information df_val a_value))
(noun: input (input df_val a_value))
(noun: interrupt (interrupt temporary-halt break df_opn
                 ts_event df_val a_value))
(noun: interval (interval ts_interval))
(noun: line (line a_component))
(noun: map (map correspondence mentob df_opn struct_obj
           df_val a_value))
(noun: mark (mark indicator-of-begin-or-end df_val a_value))
(noun: marker (marker indicator mark df_val a_value))
(noun: match (match equal df_opn df_val a_value))
(noun: measure (measure mentob))
(noun: memory memories (storage a_component))
(noun: multiplexer (multiplexer a_component))
(noun: operation (operation df_opn))
(noun: output (output df_val a_value))
(noun: parity nil (parity code_bit df_val a_value))
(noun: pause (pause delay halt ts_interval mentob))
(noun: peripheral (peripheral a_component))
(noun: phase (phase ts_interval))
(noun: point (point mark decimal-point character ts_event
             mentob))
(noun: preset (preset process-of-presetting a_component))
(noun: priority priorities ( priority privelege mentob))
(noun: process processes (process df_opn))
(noun: product (product result df_val nary_fnc df_opn))
(noun: propagation (propagation spread mentob))
(noun: quotient (quotient result df_val binary_fnc df_opn))
(noun: read (read process-of-reading df_opn ts_interval))
```

```
(noun: receipt (receipt act_of_recieving))
(noun: receiver (receiver a_component))
(noun: record (record entity_logical struct_obj df_val a_value))
(noun: register (register a_component))
(noun: report (report collection-of-facts mentob struct_obj
               df_val a_value))
(noun: request (request ask_for df_val a_value mentob))
(noun: requestor (requestor a_component))
(noun: reset (reset process-of-resetting a_component))
(noun: resolution nil (resolution mentob))
(noun: resource (resource a_component))
(noun: root (root result df_val a_value unary_fnc df_opn))
(noun: sender (sender a_component))
(noun: sequence (sequence series_of))
(noun: shift (shift df_opn))
(noun: shifter (shifter component-for-shifting a_component))
(noun: signal (signal df_val a_value))
(noun: sine (sine result df_val a_value unary_fnc df_opn))
(noun: sort (sort process-of-ordering df_opn))
(noun: speed (speed magnitude_velocity))
(noun: square (square result df_val a_value unary_fnc df_opn))
(noun: stack (stack list-data-structure struct_obj a_component))
(noun: start (start ts_event))
(noun: store (store memory a_component))
(noun: switch (switch branch_point a_component))
(noun: subsystem (subsystem part-of-system a_component))
(noun: sum (sum result df_val nary_fnc df_opn))
(noun: system (system a_component))
(noun: tag (tag label indicator df_val a_value))
(noun: tangent (tangent result df_val a_value unary_fnc df_opn))
(noun: task (task basic-work-unit activity df_opn))
(noun: test (test means-of-discrimination df_opn))
(noun: transfer (transfer vtrans process ts_event df_opn))
(noun: trap (trap special-interrupt interrupt df_opn))
(noun: unit (unit a_component))
(noun: use (use make-use-of mentob))
(noun: value (value df_val a_value))
(noun: wire (wire a_component))
(noun: word (word ordered_bits struct_obj df_val a_value))
(noun: write (write operation-of-writing df_opn ts_interval))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;VERBS
;
(verb: accept accepted accepting)
(verb: access accessed accessing)
(verb: accompany accompanied accompanying)
(verb: acknowledge acknowledged acknowledging)
(verb: activate activated activating)
(verb: address addressed addressing)
(verb: assert asserted asserting)
(verb: begin began begun beginning)
(verb: block blocked blocking)
(verb: cache cached caching)
(verb: call called calling)
(verb: channel channeled channeling)
(verb: check checked checking)
(verb: clear cleared clearing)
(verb: close closed closing)
(verb: communicate communicated communicating)
(verb: compute computed computing)
(verb: condition conditioned conditioning)
(verb: control controlled controlling)
(verb: cycle cycled cycling)
(verb: drop dropped dropping)
(verb: end ended ending)
(verb: evoke evoked evoking)
(verb: exceed exceeded exceeding)
(verb: fetch fetched fetching)
(verb: flag flagged flagging)
(verb: filter filtered filtering)
(verb: form formed forming)
(verb: format formatted formatting)
(verb: gate gated gating)
(verb: index indexed indexing)
(verb: indicate indicated indicating)
(verb: initiate initiated initiating)
(verb: input inputted inputting)
(verb: intend intended intending)
(verb: interrupt interrupted interrupting)
(verb: map mapped mapping)
(verb: mark marked marking)
(verb: match matched matching)
(verb: need needed needing)
(verb: occur occurred occurring)
(verb: pause paused pausing)
(verb: point pointed pointing)
(verb: preset preset presetting)
```

```
(verb: process processed processing)
(verb: read read reading)
(verb: receive received receiving)
(verb: record recorded recording)
(verb: report reported reporting)
(verb: request requested requesting)
(verb: reset reset resetting)
(verb: sample sampled sampling)
(verb: send sent sending)
(verb: set set setting)
(verb: share shared sharing)
(verb: shift shifted shifting)
(verb: sort sorted sorting)
(verb: start started starting)
(verb: stop stopped stopping)
(verb: store stored storing)
(verb: switch switched switching)
(verb: tag tagged tagging)
(verb: terminate terminated terminating)
(verb: test tested testing)
(verb: transfer transferred transferring)
(verb: transition transitioned transitioning)
(verb: transmit transmitted transmitting)
(verb: trap trapped trapping)
(verb: use used using)
(verb: write wrote written)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;VERB patterns
;



(index-under-pattern  (for)
 [(nil
  [(* and for)(and (or (p-o-s sentence) (and (p-o-s noun-phrase)
                (not-verb next 'basic))) (df_val))]
   [p-o-s 'adverb
    modifies-if (memq (car concept) '(uni_dir_vtrans ))
    modified-concept '( dsubst '(info (df_val ,(value 2)))
                               '(info (df_val *unspecified*))
                                (old concept))])])
```

```
(index-under-pattern  (in)
 [(nil
  [ (* and in) (and (or (p-o-s sentence) (and (p-o-s noun-phrase)
                (not-verb next 'basic)))  (a_component))]
   [p-o-s 'adverb
    modifies-if (memq (car concept) '(uni_dir_vtrans ))
 modified-concept '( dsubst '(source (a_component ,(value 2)))
                            '(source (a_component *unspecified*))
                             (old concept))])])


(name %access
      ((active passive)
       [(or (a_component) (df_opn)) (root access)
            (or (a_component) (df_opn))]
       [concept '(uni_dir_vtrans
                            (source (a_component ?a_component_1))
                            (sink   (a_component ?actor))
                            (info   (df_val ?for))
                            (control (a_component ?actor)))
       actor '?subject
       a_component_1 (value 3)
       actor    (default '*unspecified*)
       for      (default '*unspecified*)
       a_component_1 (default '*unspecified*)])


      ((active passive)
       [(or (a_component) (df_opn)) (root access) (df_val)]
       [concept '(uni_dir_vtrans
                            (source (a_component ?in))
                            (sink   (a_component ?actor))
                            (info   (df_val ?df_val_1))
                            (control (a_component ?actor)))
       actor '?subject
       df_val_1 (value 3)
       actor    (default '*unspecified*)
       in      (default '*unspecified*)
       df_val_1 (default '*unspecified*)]))


(index-under-pattern accesses
 [p-o-s 'verb])
```

```
(index-under-pattern  (accesses)
 [(nil
  [(* and accesses)]
  [p-o-s 'verb
   root 'access
   form 'basic
   number 'singular
   tense 'present
   voice 'active])])


(name %communicate
      ((active)
      [(or (a_component) (df_opn)) (root communicate) ]
      [concept '(bi_dir_vtrans
                         (src/snk_1 (a_component ?actor))
                         (src/snk_2 (a_component ?with))
                         (info    (df_val ?df_val))
                         (control (a_component ?a_component_1)))
       actor '?subject
       actor  (default '*unspecified*)
       to (default '*unspecified*)
       with   (default '*unspecified*)
       df_val (default '*unspecified*)
       a_component_1 (default '*unspecified*)])

      ((active passive)
      [(or (a_component) (df_opn)) (root communicate) (df_val)]
       [concept '(bi_dir_vtrans
                         (src/snk_1 (a_component ?actor))
                         (src/snk_2 (a_component ?to))
                         (info    (df_val ?df_val))
                         (control (a_component ?a_component_1)))
       actor '?subject
       actor  (default '*unspecified*)
       to (default '*unspecified*)
       with   (default '*unspecified*)
       df_val (default '*unspecified*)
       a_component_1 (default '*unspecified*)]))
```

```
(index-under-pattern (unary_fnc)
 [(nil
 [  (p-o-s article)
    (* and (and (or (p-o-s noun) (p-o-s noun-phrase))
                (unary_fnc))) of (pname)]
 [ p-o-s 'noun-phrase
   description '(fnc_val)
   number (value 2 number)
cd-form '(fnc_info
          (input1(a_value ,(new-token '(a_value)(value 4 word)))
          (output1 (a_value *unspecified*))
          (operation(df_opn ,(new-token '(df_opn)(value 2 word)))
                           (arity 1)))])
 (nil
 [(p-o-s article)
  (* and (and (or (p-o-s noun) (p-o-s noun-phrase)) (unary_fnc)))
                      (pname) of (pname)]
 [ p-o-s 'noun-phrase
   description '(fnc_val)
   number (value 2 number)
cd-form '(fnc_info
        (input1 (a_value ,(new-token '(a_value)(value 5 word)))
        (output1 (a_value ,(new-token '(a_value)(value 3 word)))
    .   (operation (df_opn ,(new-token '(df_opn)(value 2 word)))
                              (arity 1)))])])


(index-under-pattern (binary_fnc)
 [(nil
 [  (p-o-s article)
    (* and (and (or (p-o-s noun) (p-o-s noun-phrase))
                (binary_fnc))) of (pname) and (pname)]
 [ p-o-s 'noun-phrase
   description '(fnc_val)
   number (value 2 number)
cd-form '(fnc_info
         (input1 (a_value ,(new-token '(a_value)(value 4 word)))
         (input2 (a_value ,(new-token '(a_value)(value 6 word)))
         (output1 (a_value *unspecified*))
         (operation (df_opn ,(new-token '(df_opn)(value 2 word)))
                              (arity 2))
                )])
```

```
 (nil
 [  (pname) %comma (p-o-s article)
    (* and (and (or (p-o-s noun) (p-o-s noun-phrase))
           (binary_fnc))) of (pname) and (pname)]
 [ p-o-s 'noun-phrase
   description '(fnc_val)
   number (value 4 number)
cd-form '(fnc_info
         (input1 (a_value ,(new-token '(a_value)(value 5 word))))
         (input2 (a_value ,(new-token '(a_value)(value 7 word))))
         (output1(a_value ,(new-token '(a_value)(value 1 word))))
         (operation (df_opn ,(new-token '(df_opn)(value 4 word)))
                              (arity 2))
               )])])


(name %compute
      ((active passive)
      [  (a_component) (root compute) (fnc_val)]

      [concept '(non_dir_vtrans
                        (fnc_info ?fnc_info)
                      (m_b_n_r (df_opn ?df_opn)
                             (a_component ?actor)
                             (ts_interval ?ts_interval)))

       actor '?subject
       fnc_info (cdr (value 3 cd-form))
       df_opn  (cadr (assoc 'operation (cdr (value 3 cd-form))))
       actor (default '*unspecified*)
       ts_interval (default '*unspecified*)]))


(index-under-pattern  (to)
 [(nil
  [(* and to)(and (or (p-o-s sentence) (and (p-o-s noun-phrase)
               (not-verb next 'basic))) (or (a_component)
               (df_opn)))]
   [p-o-s 'adverb
    modifies-if (memq (car concept) '(uni_dir_vtrans ))
 modified-concept '( dsubst '(sink (a_component ,(value 2)))
                           '(sink (a_component *unspecified*))
                            (old concept))])])
```

```
(index-under-pattern  (from)
 [(nil
  [(* and from)(and (or (p-o-s sentence)(and (p-o-s noun-phrase)
                   (not-verb next 'basic))  (or (a_component)
                   (df_opn)))]
   [p-o-s 'adverb
    modifies-if (memq (car concept) '(uni_dir_vtrans ))
 modified-concept '( dsubst '(source (a_component ,(value 2)))
                            '(source (a_component *unspecified*))
                             (old concept))])])


(name %send
      ((active passive)
        [ (or (a_component) (df_opn)) (root send) (df_val)]
        [concept '(uni_dir_vtrans
                            (source (a_component ?from))
                            (sink   (a_component ?to))
                            (info   (df_val ?df_val))
                            (control (a_component ?actor)))
       actor '?subject
       df_val (value 3)
       actor (default '*unspecified*)
       to   (default '*unspecified*)
       from (default '*unspecified*)
       df_val (default '*unspecified*)])

      ((active )
        [ (or (a_component) (df_opn)) (root send)
                            (or (a_component) (df_opn)) (df_val)]
       [concept '(uni_dir_vtrans
                            (source (a_component ?from))
                            (sink   (a_component ?to))
                            (info   (df_val ?df_val))
                            (control (a_component ?actor)))
       actor '?subject
       df_val (value 4)
       actor (default '*unspecified*)
       to   (or (value 3) (default '*unspecified*))
       from (default '*unspecified*)
       df_val (default '*unspecified*)]))
```

```
(name %transfer
      ((active passive)
       [(or (a_component) (df_opn)) (root transfer) (df_val)]
       [concept '(uni_dir_vtrans
                            (source (a_component ?from))
                            (sink   (a_component ?to))
                            (info   (df_val ?df_val))
                            (control (a_component ?actor)))
       actor  '?subject
       df_val (value 3)
       to     (default '*unspecified*)
       from   (default '*unspecified*)
       df_val (default '*unspecified*)]))

(name %begin
      ((active)
    [(or (a_component) (df_opn) (ts_interval))
           (and (root begin) (not-noun next) (not-article next))]
       [concept '(single_temporal_event
                     (ts_event_init_1 (event_name ?actor)
                                      (event_type ?event_type)))

        actor  '?subject
        event_type_1 (default '*pi*)])

      ((active passive)
      [(or (a_component)(df_opn)(ts_interval)) (root begin)
                   (or (a_component)(df_opn)(ts_interval))]
 [concept '(causal_temporal_init
              (ts_event_init_1 (event_name ?actor)
                               (event_type ?event_type_init_1))
              (ts_arc_1        (arc_type ?arc_type_1)
                               (arc_tail ?actor)
                               (arc_head ?event_name_caus_1)
                               (arc_rel ?arc_rel_1)
                               (arc_len ?arc_len_1)
                               (arc_units ?arc_units_1))
              (ts_event_caus_1 (event_name ?event_name_caus_1)
                               (event_type ?event_type_caus_1))
              (ts_event_init_2 (event_name ?a_component_2)
                               (event_type ?event_type_init_2))
              (ts_arc_2        (arc_type ?arc_type_2)
                               (arc_tail ?arc_tail_2)
                               (arc_head ?a_component_2)
                               (arc_rel ?arc_rel_2)
                               (arc_len  ?arc_len_2)
                               (arc_units ?arc_units_2)))
```

```
        actor  '?subject
        actor (default '*unspecified*)
        event_type_init_1 (default '*pi*)
        arc_type_1 (default '*sigma*)
        arc_rel_1 (default 'equal)
        arc_len_1 (default '*unspecified*)
        arc_units_1 (default 'seconds)
        event_name_caus_1 (default '*caus*)
        event_type_caus_1 (default '*beta*)
        event_type_init_2 (default '*pi*)
        arc_type_2 (default '*causal*)
        arc_tail_2 (default '*caus*)
        arc_rel_2 (default 'equal)
        arc_len_2 (default 0)
        arc_units_2 (default 'seconds)
        a_component_2 (value 3)
        a_component_2 (default '*unspecified*)]))


(name %terminate
      ((active)
    [(or (a_component) (df_opn) (ts_interval))
      (and (root terminate) (not-noun next) (not-article next))]
       [concept '(single_temporal_event
                      (ts_event_term_1 (event_name ?actor)
                                       (event_type ?event_type)))

        actor  '?subject
        event_type (default '*pi*)])
      ((active passive)
      [(or (a_component) (df_opn) (ts_interval)) (root terminate)
                          (or (a_component)(df_opn)(ts_interval))]
[concept '(causal_temporal_term
              (ts_event_init_1 (event_name ?actor)
                               (event_type ?event_type_init_1))
              (ts_arc_1        (arc_type ?arc_type_1)
                               (arc_tail ?actor)
                               (arc_head ?event_name_caus_1)
                               (arc_rel ?arc_rel_1)
                               (arc_len ?arc_len_1)
                               (arc_units ?arc_units_1))
             (ts_event_inter_1 (event_name ?actor)
                               (event_type ?event_type_inter_1))
```

```
             (ts_event_init_2 (event_name ?a_component_2)
                              (event_type ?event_type_init_2))

             (ts_arc_2       (arc_type ?arc_type_2)
                             (arc_tail ?a_component_2)
                             (arc_head ?a_component_2)
                             (arc_rel ?arc_rel_2)
                             (arc_len  ?arc_len_2)
                             (arc_units ?arc_units_2))
            (ts_event_inter_2 (event_name ?a_component_2)
                              (event_type ?event_type_inter_2))
             (asynch_pred     (predicate *unspecified*)
                              (a_component *unspecified*)
                              (ts_interval ?ts_event_init_1)
                          (ts_destination (event_name ?terminal)
                                          (event_type *rho*))))
       actor  '?subject
       actor (default '*unspecified*)
       event_type_init_1 (default '*pi*)
       arc_type_1 (default '*sigma*)
       arc_rel_1 (default 'equal)
       arc_len_1 (default '*unspecified*)
       arc_units_1 (default 'seconds)
       event_name_inter_1 (default '*unspecified*)
       event_type_inter_1 (default '*pi*)
       event_type_init_2 (default '*pi*)
       event_type_inter_2 (default '*pi*)
       arc_type_2 (default '*sigma*)
       arc_tail_2 (default '*unspecified*)
       arc_rel_2 (default 'equal)
       arc_len_2 (default '*unspecified*)
       arc_units_2 (default 'seconds)
       a_component_2 (value 3)
       a_component_2 (default '*unspecified*)
       terminal (default '*unspecified*)]))


(index-under-pattern before
 [p-o-s 'conjunction])

(index-under-pattern  ( before)
 [(nil
  [ (* and before) (or (p-o-s sentence)
                      (and (p-o-s noun-phrase)
                           (not-verb next 'basic)))
                    ([(%comma)])]
    [p-o-s 'adverb
```

```
   modifies-if (memq (car concept) '(uni_dir_vtrans bi_dir_vtrans
                                     single_temporal_event
                                     causal_temporal_init
                                     causal_temporal_term))
    modified-concept '(append1 (old concept)
         '(single_temporal_event
          (ts_arc_constraint  (arc_type ,(default '*constraint*))
                              (arc_head ,(default '*succ*))
                              (arc_tail ,(default '*pred*))
                              (arc_rel  ,(default 'gt))
                              (arc_len  ,(default 0))
                              (arc_units ,(default 'seconds)))
                             (*succ*  ,(value 2 cd-form))))])])

(index-under-pattern  ( ts_measure before)
 [(nil
   [ (ts_measure) (* and before) (or (p-o-s sentence)
                                     (and (p-o-s noun-phrase)
                                          (not-verb next 'basic)))
                                      ([(%comma)])]

   [p-o-s 'adverb
   modifies-if (memq (car concept) '(uni_dir_vtrans bi_dir_vtrans
                                 single_temporal_event
                                 causal_temporal_init
                                 causal_temporal_term))
  modified-concept '(append1 (old concept)
          '(single_temporal_rel
          (ts_arc_constraint  (arc_type ,(default '*constraint*))
                                (arc_head ,(default '*succ*))
                                (arc_tail ,(default '*pred*))
                                (arc_rel  ,(or (value 1 relation)
                                               (default 'gt)))
                                (arc_len  ,(or (value 1 amount)
                                               (default 0)))
                                (arc_units ,(or (value 1 units)
                                           (default 'seconds))))
                           (*succ*  ,(value 3 cd-form))))])])

(index-under-pattern during
 [p-o-s 'conjunction])

(index-under-pattern  (during)
   [(nil
   [ (* and during) (or (p-o-s sentence)
                        (and (p-o-s noun-phrase)
                             (not-verb next 'basic)))
                        ([(%comma)])]
```

```
    [p-o-s 'adverb
   modifies-if (memq (car concept) '(uni_dir_vtrans bi_dir_vtrans
                                     single_temporal_event
                                     causal_temporal_init
                                     causal_temporal_term))
  modified-concept '(append1 (old concept)
            '(dual_temporal_rel
             (ts_event_init_1 (event_name
                               ,(or (value 2) (value 2 word)
                                   (default '*unspecified*)))
                              (event_type ,(default '*beta*)))
             (ts_event_term_1 (event_name
                               ,(or (value 2) (value 2 word)
                                    (default '*unspecified*)))
                              (event_type ,(default '*mu*)))
             (ts_arc_1        (arc_type ,(default '*constraint*))
                              (arc_tail ,(default '*beta*))
                              (arc_head ,(default '*pred-init*))
                              (arc_rel ,(default 'gt))
                              (arc_len ,(default 0)))
             (ts_arc_2        (arc_type ,(default '*constraint*))
                              (arc_tail ,(default '*pred-term*))
                              (arc_head  ,(default '*mu*))
                              (arc_rel ,(default 'gt))
                              (arc_len  ,(default 0))))))]])


(index-under-pattern (on)
 [(nil
  [(* and on) ( a_component)]
  [p-o-s 'adverb
  modifies-if (memq (car concept) '(uni_dir_vtrans bi_dir_vtrans
                                    single_temporal_event
                                    causal_temporal_init
                                    causal_temporal_term))
 modified-concept '(append1 (old concept)
                  '(v_c_n_r  (df_val *unspecified*)
                     (a_component ,( or (value 2)(value 2 word)))
                            (ts_interval *unspecified*)))]])
```

```
(index-under-pattern  (noun-phrase and noun-phrase)
 [(nil
  [ %comma (p-o-s noun-phrase)
          and (* and (p-o-s noun-phrase)
              (or (not-verb next 'basic) (e (eq next '%end%))))
                ([%comma])]
[p-o-s 'appositive
  cd-form '(supplementary-concepts appositive)
do (add-relation-to-*sc* 'ref (value 2 cd-form)
                              (value 2 description))
do (add-relation-to-*sc* 'ref (value 4 cd-form)
                              (value 4 description)])])


(index-under-pattern (decl_pname)              ; jjg 14 sep 1986
 [(nil                                          ; for use of
 [ (* and (p-o-s noun))]                        ; unqualified pnames
 [p-o-s 'noun-phrase
    description (value 1 description)           ; that are declared
    cd-form (value 1 cd-form)
    do (copy-term 1)])])


; The next pattern actually handles special noun noun cases but
;must be indexed under noun, to allow look ahead -- the result
;is a noun phrase that contains one noun.

(index-under-pattern (noun)
 [(nil
 [ (or (p-o-s article) (p-o-s quantifier))                ; Heuristic
        (* and (and (p-o-s noun)                           ; a noun
                                                           ; next (or
                     (noun-and-verb next)   ; a verb with the
                     (not (not-plural next)); same stem) and
                (not-verb after-next 'basic)))] ; plural in number
                                                ; and there isn't
                                                ; a verb after
                                                ; this noun
   [ p-o-s 'noun-phrase
     ref (value 1 ref)
     class (value 2 word)
     cd-form (old-token (value 2 description) (value 2 word))
     description (value 2 description)
     do (add-adjs-to-*sc* (value 1 adjs)
                          (terms cd-form))
     do (copy-term 3)])])
```

```
; The next pattern actually handles special noun noun cases but
;must be indexed under noun, to allow look ahead -- the result
is an ambiguous sentence.


(index-under-pattern (noun)
 [(nil
 [ (or (p-o-s article) (p-o-s quantifier))        ; Heuristic 2
        (* and (and (p-o-s noun)                    ; a noun-and-verb
                          (noun-and-verb next)    ; followed by a
                         (not (not-plural next)); second
                              (noun-and-verb after-next))
             (p-o-s noun) (p-o-s noun)]            ; noun-and-verb
                                                   ;
                                                   ;
                                                   ;
   [ p-o-s 'ambiguous-sentence
     do (add-to-*sc* '(ambiguous-sentence (verb1 ,(value 3 word))
                                          (verb2 ,(value 4 word))))])])



(index-under-pattern (adjective noun )
 [(nil
   [ (or (p-o-s article) (p-o-s quantifier)) (p-o-s adjective)
                    (* and (and ( p-o-s noun) (not-noun next)))]
   [ p-o-s 'noun-phrase
     cd-form
    (old-token (value 3 description) (value 3 word))
     description (value 3 description)
     do (add-adjs-to-*sc* (value 2 adjs)
                          (terms cd-form))
     do (copy-term 4)])])


(index-under-pattern (verb noun )
 [(nil
  [(or (p-o-s article) (p-o-s quantifier))
     (and  (p-o-s verb)(or (form perfective) (form progressive)))
                    (* and (and ( p-o-s noun) (not-noun next)))]
   [ p-o-s 'noun-phrase
     cd-form
   (old-token (value 3 description)
              (atcat (value 2 word) '\- (value 3 word)))
     description (value 3 description)
     do (copy-term 4)])])
```

```
; The next pattern actually handles special noun noun cases but
; must be indexed under noun, to allow look ahead--the result
; is a noun phrase that contains one noun.

(index-under-pattern (adjective noun )
 [(nil
   [ (or (p-o-s article) (p-o-s quantifier))(p-o-s adjective)
                                                     ; Heuristic
      (* and (and (p-o-s noun) (not (not-noun next)); a noun next
                                                     ; (or a verb
                             (noun-and-verb next)    ; with the
                             (not (not-plural next)); same stem)
                                                     ; and plural
                    (not-verb after-next 'basic)))]  ; in number
                                                     ; and there
                                                     ; isn't a
                                                     ; verb after
                                                     ; this noun
   [ p-o-s 'noun-phrase
     cd-form
    (old-token (value 3 description) (value 3 word))
     description (value 3 description)
     do (add-adjs-to-*sc* (value 2 adjs)
                           (terms cd-form))
     do (copy-term 4)])])

(index-under-pattern (number noun)
 [(nil
   [(p-o-s number) (* number plural)]
   [p-o-s 'noun-phrase
    cd-form (or (value 2)
                (new-token (value 2 description)
                           (value 2 word)))
    description (value 2 description)
    do (add-to-*sc* '(number (group ,(terms cd-form))
                             (number ,(value 1))))])])
```

```
(index-under-pattern ( number adjective noun )
 [(nil
   [ (or (p-o-s article) (p-o-s quantifier))
     (p-o-s number) (p-o-s adjective)
     (* and (and ( p-o-s noun) (number plural)
                 (not-noun next)))]
   [ p-o-s 'noun-phrase
     cd-form
    (old-token (value 4 description)
               (old-token (value 4 description)
                          (value 4 word)))
     description (value 4 description)
     do (add-to-*sc* '(number (group ,(terms cd-form))
                              (number ,(value 2))))
     do (add-adjs-to-*sc* (value 3 adjs)
                          (terms cd-form))
     do (copy-term 5)])])

; The next pattern actually handles special noun noun cases but
; must be indexed under noun, to allow look ahead -- the result
; is a noun phrase that contains one noun.


(index-under-pattern ( number adjective noun )
 [(nil
   [ (or (p-o-s article) (p-o-s quantifier))
     (p-o-s number)(p-o-s adjective
                                                   ; Heuristic
     (* and (and (p-o-s noun) (number plural)
                              (not (not-noun next)); a noun next
                              (noun-and-verb next) ; (or a verb
                                                   ; with the
                            (not (not-plural next)); same stem)
                                                   ; and plural
                   (not-verb after-next 'basic)))] ; number and
                                                   ; there isn't
                                                   ; a verb after
                                                   ; this noun
   [ p-o-s 'noun-phrase
     cd-form
    (old-token (value 4 description)
               (old-token (value 4 description) (value 4 word)))
     description (value 4 description)
     do (add-to-*sc* '(number (group ,(terms cd-form))
                              (number ,(value 2))))
     do (add-adjs-to-*sc* (value 3 adjs)
                          (terms cd-form))
     do (copy-term 5)])])
```

```
(index-under-pattern (noun noun)
 [(nil
  [(or(p-o-s article)(p-o-s quantifier)) (p-o-s noun)
                     (* and (and ( p-o-s noun) (not-noun next)))]
   [ p-o-s 'noun-phrase
     ref (value 1 ref)
     class (value 3 word)
     cd-form
   (old-token (value 3 description)
              (atcat (value 2 word) '\- (value 3 word)))
     description (value 3 description)
     do (add-adjs-to-*sc* (value 1 adjs)
                          (terms cd-form))
     do (copy-term 3)])])

(index-under-pattern one
 [p-o-s 'number
  cd-form 1])

(index-under-pattern (number one)
 [(nil
  [(p-o-s number) (* and one)]
  [p-o-s 'number
   cd-form (eval (plus (value 1) (value 2)))])])

(index-under-pattern tenths
 [p-o-s 'number
  cd-form .1])

(index-under-pattern (number tenths)
 [(nil
  [(p-o-s number) (* and tenths)]
  [p-o-s 'number
   cd-form (eval (times (value 1) (value 2)))])])
                              (ts_interval *unspecified*)))]])])

(index-under-pattern (for ts_measure)
 [(nil
   [ for (* ts_measure)]
   [p-o-s 'adverb
 modifies-if (memq (car concept) '(uni_dir_vtrans bi_dir_vtrans))
 modified-concept '(append1 (old concept)
            '(single_temporal_rel (ts_constraint (value 2))))])])
```

```
(index-under-pattern (number ts_units)
 [(nil
 [ (p-o-s number) (* ts_units)]
  [p-o-s 'noun-phrase
   description '(ts_measure)
   cd-form '( ts_measure (relation ?relation)(amount ?amount)
                                (units ?units))
    relation (default 'equal)
   amount (eval (times (value 1) (value 2)))
   units (default 'seconds)])])


(index-under-pattern ( relation number ts_units)
 [(nil
 [ (relation) (p-o-s number) (* ts_units)]
  [p-o-s 'noun-phrase
   description '(ts_measure)
   cd-form '( ts_measure (relation ?relation)(amount ?amount)
                                (units ?units))
   relation (value 1)
   relation (default 'equal)
   amount (eval (times (value 2) (value 3)))
   units (default 'seconds)])])


(name %nanoseconds
      (nil
      [(p-o-s noun)]
      [p-o-s 'noun-phrase
       cd-form  1E-9
       description '(ts_units)
       do (copy-term 1)]))

(index-under-pattern ns
  [p-o-s 'noun])

(index-under-pattern nanoseconds
  [p-o-s 'noun])

(index-under-pattern nanosecs
  [p-o-s 'noun])

(index-under-pattern (ns)
  %nanoseconds
  ((* and 1)))
```

```
(index-under-pattern (nanoseconds)
  %nanoseconds
   ((* and 1)))

(index-under-pattern (nanosecs)
  %nanoseconds
    ((* and 1)))

(index-under-pattern less
 [p-o-s 'adjective])

(index-under-pattern than
 [p-o-s 'preposition])

(index-under-pattern (less than)
 [(nil
   [ less (* and than) ]
   [p-o-s 'adv-rel
    description '(relation)
    cd-form 'lt ])])

(index-under-pattern first
 [p-o-s 'adjective
  cd-form '(ordered starting-one)
  adjs '((ordered starting-one))
  state 'ordered
  val 'starting-one])


(index-under-pattern (pname is noun-phrase)
 [(nil
   [(pname) (root be) (* p-o-s noun-phrase]
   [p-o-s 'sentence
    cd-form  '(declaration
                     (pname ?descriptor)
                     (ref ?ref)
                     (class ?class)
                     (description ?nominative))
     descriptor  (value 1 word)
     ref (value 3 ref)
     class (value 3 class)
     nominative  (value 3 description)])])
```

```
(index-under-pattern (verb)   ; imperative --- this pattern is
                              ; also necessary so that adverbs
 (get (from-end 1 root))      ; work correctly when the
 [(* and 2 (beginning) (negative nil)) %rest
                              ; adverb occurs first
                              ; jjg 24 sep 86
    (and %last (e (neq next '%question-mark%)))]
                              ; doesn't work when
                              ; %last is optional!
 (subject '*you*
  cd-form '?concept
  imperative t
  p-o-s 'sentence) active)
```