Mapping Conceptual Graphs to Primitive VHDL Processes

by

Vikram M. Shrivastava

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Electrical Engineering

APPROVED:

[illegible]

Dr. J. R. Armstrong, Chairman

[illegible]

Dr. W. R. Cyre

[illegible]

Dr. F. G. Gray

June 1994

Blacksburg, Virginia

C.2

Mapping Conceptual Graphs to Primitive VHDL Processes

by

Vikram M. Shrivastava

Dr. James R. Armstrong, Chairman

Electrical Engineering

(ABSTRACT)

This thesis discusses an algorithm for mapping conceptual graphs to primitive VHDL processes. The behavior of each primitive process is stored in the form of a schema. The algorithm identifies concepts in the input referring to MODAS (Modeler's Assistant) process primitives and maps their schemata to the input conceptual graph. The results of the mapping are used to modify the primitive process's VHDL and instantiate a new process. A library of schemata for the primitive processes in MODAS has been developed.

This algorithm has been implemented in the CGVHDL Linker program. It has improved the capability of the CGVHDL Linker to handle more complex design specifications. The algorithm provides the CGVHDL Linker with an ability to interpret a structure in the input conceptual graph. It also eases the burden on the designer who can refer to some components without giving details of their behavior.

Acknowledgments

I would like to thank my advisor Dr. James R. Armstrong for support and guidance in my research. It has been a pleasure and a privilege to work with him. I also thank Dr. W. R. Cyre and Dr. F. G. Gray for serving as members of my graduate committee.

I would like to thank a lot of friends who have provided me with inspiration and encouragement in all my endeavors. I am grateful to Sanat Rao and Shekhar Kapoor for all their advice and motivation. I appreciate the company of Anand Joshi, David Dailey Chang Cho, John Wicks, Sathya, Prasad, Rekha, Hormuzd, Al Walters(Snowman), Thorn and many others.

I dedicate this work to my parents who have provided encouragement and inspiration over the difficult times. I also thank my brother Rohit Srivastava for all his support.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Over the years the integrated circuit industry has been making great strides in fabrication and design technology. The design of microprocessors and ASIC's is constantly increasing in complexity and performance. Electronic Design Automation plays a big role in the successful design of complex VLSI chips. Developments in EDA have produced CAD tools for simulating a system design, system analysis and design verification. To facilitate the design of systems, the use of conventional languages such as C and PASCAL was replaced by *hardware description languages*. This was followed by setting up HDL standards and the IEEE 1076 standard was accepted. The commonly used HDL is the VHSIC hardware description language or VHDL.

VHDL has a specialized set of language constructs for accurate modeling of digital systems. Designs can be described at various levels of abstraction. VHDL modeling provides a designer with various styles like algorithmic, data flow or structural. A design process starts with the development of a VHDL model, analyzing or compiling the model, followed by model simulation. The simulation output is studied for verifying the functionality of the model. A designer can then synthesize the circuit and have it placed and routed on a chip as the final step.

An important consideration in EDA development is its interface to the designer. Developing and testing HDL models of digital circuits is a laborious and time consuming task. This is the motivation for some of the research work at the Design Automation Laboratory, Bradley Department of Electrical Engineering, Virginia Tech. The ASPIN project aims to ease the task and reduce the time of model development. Using the ASPIN system, the designer can provide digital circuit information at a higher level of abstraction. The ASPIN system provides the user with an interface for a number of forms of informal specifications, i.e., natural language, block diagrams and flow charts.

The ASPIN system first translates the information in the specification to a form of knowledge representation called *conceptual graphs*. The conceptual graphs generated by all these interfaces are then combined to form a single specification graph. This graph is then interpreted to generate the HDL specification by the final stage of the ASPIN system, i.e., Conceptual Graph to VHDL Linker or CGVHDL Linker. The CGVHDL Linker generates output files which can be read by another application program called the Modeler's Assistant. The Modeler's Assistant can display the model in the form of a process model graph and can generate its VHDL file.

1.2 Contributions

The CGVHDL Linker has undergone two stages of development. The first version of the CGVHDL Linker identified action and behavior *concepts* in a conceptual graph and mapped them to VHDL constructs *[Honcharik 93]*. The contributions discussed in this thesis have the following salient features,

(i) The CGVHDL Linker can interpret a structure in the conceptual graph. The user can provide a specification referring to certain VHDL primitives and the interconnection between them. This makes the task of the designer easier as he does not specify detailed behavior of each primitive.

(ii) A graph matching algorithm is used to map a schema onto the input conceptual graph to form a subgraph which refers to the primitive VHDL process represented by the schema.

(iii) A library of schemata representing the behavior of the primitive processes in the MODAS primitive library has been developed.

1.2 Contents

Chapter 2, "ASPIN and MODAS" introduces the ASPIN system and the Modeler's Assistant which are the other key CAD tools used in this work..

Chapter 3, "A Conceptual Graph Primer" is a simple tutorial on conceptual graphs and various terms involving them that occur in the following chapters.

Chapter 4, "The Projection Operator" discusses the heart of the new algorithm, i.e., the process of mapping a schema to the input conceptual graph called *projection*. Various schemes for graph matching are reviewed.

Chapter 5, "CGVHDL Linker: The projection based algorithm" enumerates the various steps involved in the new algorithm for the CGVHDL Linker. This chapter describes the algorithm along with a simple example to show results available after each stage.

Chapter 6, "Conclusion" reviews the success and achievements of the CGVHDL Linker and proposes some ideas for future enhancement.

Chapter 2

ASPIN and MODAS

2.1 Introduction

To facilitate the circuit design process there are two CAD systems under development at the Design Automation Lab, Virginia Tech , i.e.,

i. The Modeler's Assistant or MODAS, and

ii. Automated Specification Interpreter or ASPIN

Both these systems currently have working versions, and are used for design work and other research. The aim of both the systems is to provide the designer with a user interface which would allow him to specify circuit details with ease. The MODAS system works at the level of VHDL input but the ASPIN system is being developed to take an input in the form of informal specifications and translate them to VHDL.

2.2 The Modeler's Assistant

The Modeler's Assistant or MODAS is a tool to ease the burden on a designer of developing large VHDL behavioral models in a small time frame. MODAS works interactively with the user to generate a pictorial representation of the architecture body of a behavioral VHDL model called the ***process model graph*** (PMG).

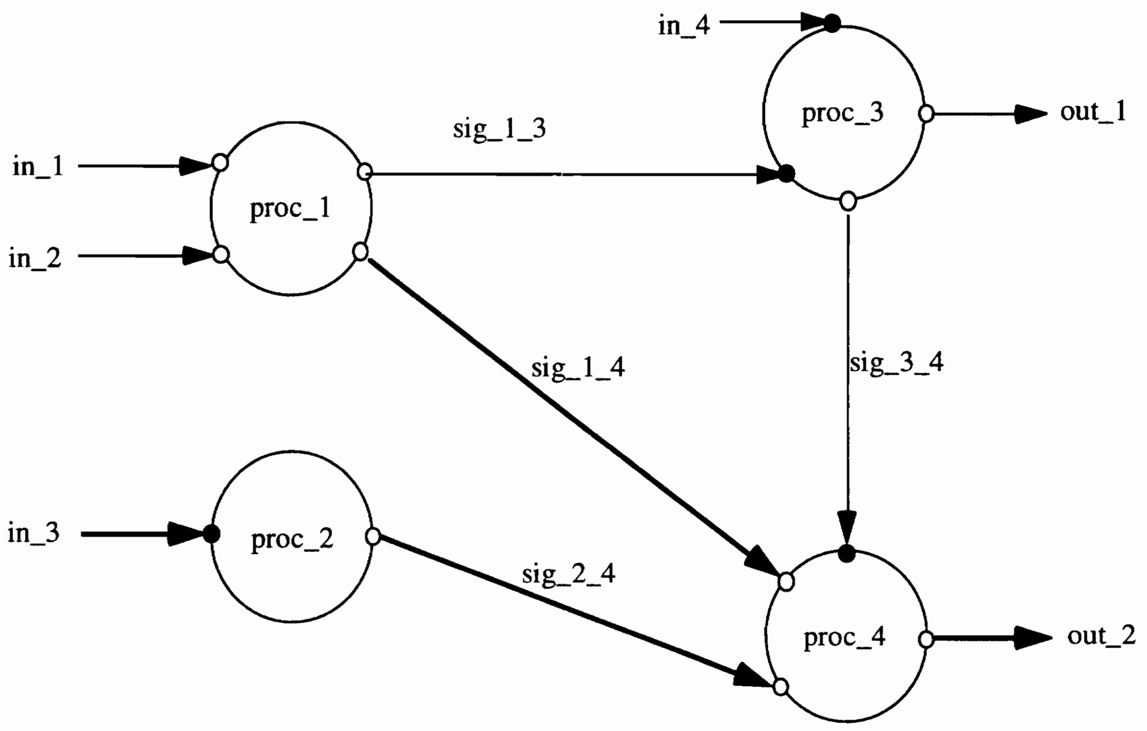


Fig. 2.1 Process Model Graph

An example of a PMG is shown in Fig. 2.1. This behavioral model has 4 processes, 4 inputs and 2 outputs, and 4 interconnecting signals between the processes. Each of the larger circles represents a process. The small circles on the periphery are inputs and outputs of the process and are called ports. The inputs to the process which are in the sensitivity list are shown by filled port circles.

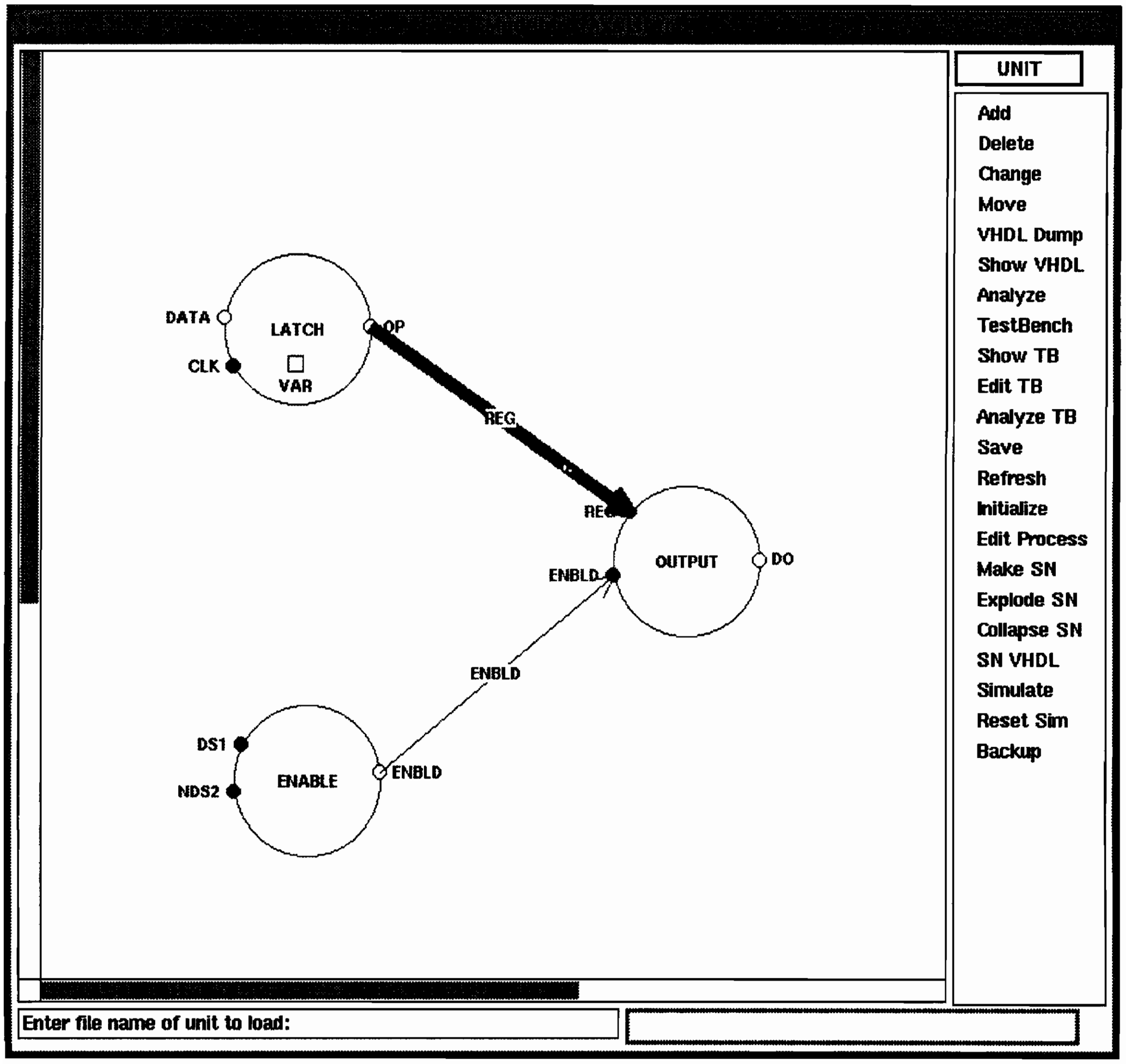


Fig. 2.2 MODAS User Interface

The designer first uses MODAS to lay out a PMG by instantiating processes. MODAS provides a library of ***primitives*** that consists of frequently used processes. A user may use a primitive to instantiate several processes of it's type with different names. The primitive provides the default VHDL code for the new process generated from it. This is a feature used by the CGVHDL Linker to obtain VHDL code of a primitive process. VHDL is specified for other processes by editing. The PMG is completed by interconnecting the process ports by using signals. The MODAS user interface is shown in Fig. 2.2. Once the PMG is complete, the user can select *VHDL Dump* from a menu and have full VHDL for the model written to a file. The VHDL code generated by MODAS for the PMG in Fig. 2.2 is shown in Fig. 2.3.

```
use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
-- *********************************************************
entity latch8 is
 port (DO: out BIT_VECTOR(0 to 7);
   NDS2: in BIT;
   DS1: in BIT;
   CLK: in BIT;
   DATA: in BIT_VECTOR(0 to 7));
end latch8;
-- *********************************************************

architecture BEHAVIORAL of latch8 is

 signal ENBLD: BIT;
 signal REG: BIT_VECTOR(0 to 7);
begin

 -- -----------------------------------------------------
 -- Process Name: OUTPUT
 -- -----------------------------------------------------

 OUTPUT_4: process (ENBLD,REG)
 begin
   if (ENBLD = '1') then
```

Fig. 2.3 VHDL code for the PMG in Fig. 2.2 (contd. ..)

```
            DO <= REG;
    else
            DO <= "00000000";
    end if;


 end process OUTPUT_4;


 -- ------------------------------------------------------
 -- Process Name: ENABLE
 -- ------------------------------------------------------

 ENABLE_9: process (NDS2,DS1)
 begin
    ENBLD <= DS1 and not NDS2;


 end process ENABLE_9;


 -- ------------------------------------------------------
 -- Process Name: LATCH
 -- ------------------------------------------------------

 LATCH_14: process (CLK)
    variable VAR: BIT_VECTOR(0 to 7);
 begin
    if (CLK = '0') then
           REG <=  DATA;
          VAR := DATA;
    else
           REG <= VAR;
    end if;


 end process LATCH_14;


end BEHAVIORAL;
```

Fig. 2.3 VHDL code for the PMG in Fig. 2.2

A feature of MODAS also allows the user to add processes as primitives to the primitive library. Apart from just a user-interface for PMG design, MODAS offers other services. MODAS can analyze and simulate VHDL code of a PMG by making system calls to Synopsys. Simulation results can be displayed with waveforms or on the PMG. MODAS is also interfaced with other tools that have been developed for testing VHDL models. MODAS can call these tools to compile tests for processes and entities. These tests are converted to a test bench by another tool.

2.3 The ASPIN system

The ASPIN system provides various user interfaces to generate VHDL code from informal specifications like natural language, block diagrams and flowcharts. A block diagram of the ASPIN system is shown in Fig, 2.4.

The Natural Language Interface is comprised of a Parser and a Semantic Analyzer. This interface parses a set of sentences and generates ***parse trees***. The parse trees are then converted to conceptual graphs using a Semantic Analyzer. The Block Diagram, Flow Chart and Timing Diagram interfaces are currently under development. Each of these employ the conceptual graph for storing the specification information. After all the information has been provided, the conceptual graphs from all these subsystems are integrated into a single conceptual graph. This is done by using the Conceptual Graph Processor. This block takes numerous graphs and performs a join between them by using a coreference detector and a set of joining rules. The Conceptual Graph Processor is still under development and for the current system only the coreference detector is functional. The final functional blocks are a set of conceptual graph interpreters which generate a

VHDL model or a graphical interpretation. The VHDL model is generated by the CGVHDL Linker along with MODAS.

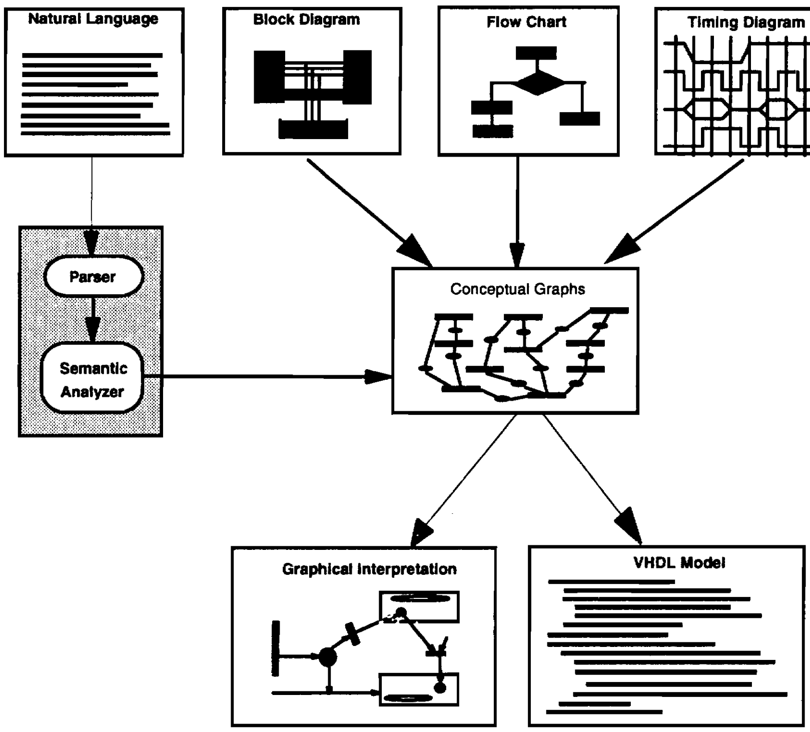


Fig. 2.4 The ASPIN System

The CGVHDL Linker generates MODAS files for various processes that can be interpreted from the integrated conceptual graph. It also generates a file for the entire entity which contains all the signal connections within the architecture body of the VHDL model. This information is read by MODAS and displayed in the form of a PMG. Using a function of MODAS the PMG is converted into a full VHDL file.

Chapter 3

A Conceptual Graph Primer

3.1 Introduction

A conceptual graph is a structure used for knowledge representation. They are formed based on logic, linguistics, psychology, and philosophy *[Sowa 84]*. A conceptual graph is an abstract form of representing the process of perception. Conceptual graphs used in this work were generated through a semantic analyzer, hence they represent information being conveyed by the sentence.

3.2 Conceptual Graphs

Definition.

*A **conceptual graph** is a finite, connected, bipartite graph. [Sowa 84]*

A conceptual graph is *finite* because any graph that is processed by humans or a machine must have a finite number of nodes. Such graphs are *connected graphs* because if some parts were not connected, they would constitute different graphs. These graphs consist of two types of nodes, i.e., concepts and conceptual relations, and each arc connects a node of one type to a node of the other type, hence they are *bipartite*.

Definition.

*A **concept** represents any entity, action or state that can be described in language.*

All devices, actions, states, events, attribute and values are types of concepts encountered in the ASPIN system.

Definition.

*A **conceptual relation** specifies the roles played by each concept on others.*

Consider a simple sentence, "The UART on the chip communicates with other devices". This sentence has a simple conceptual graph shown in Fig. 3.1.

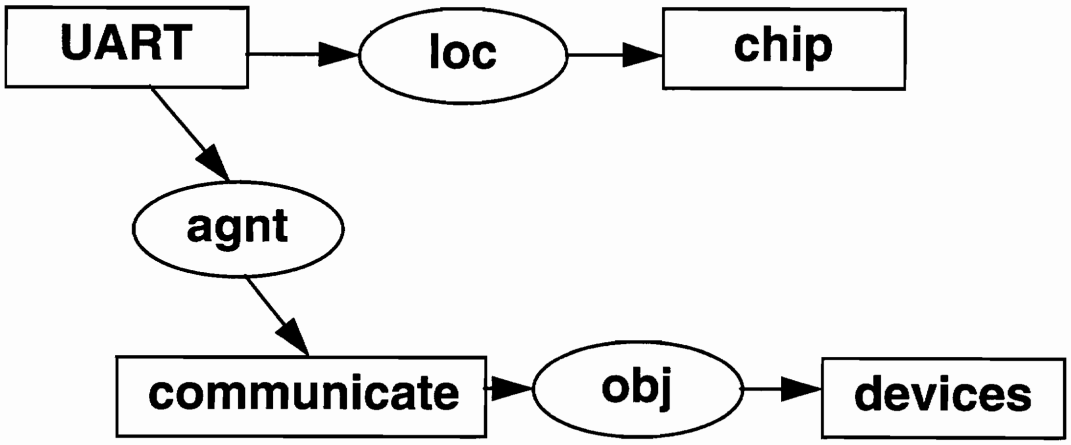


Fig. 3.1 A simple conceptual graph

In the conceptual graph shown in Fig. 3.1, the rectangular blocks are concepts and the ovals represent conceptual relations. The words within the symbols are *labels or referents* for the node. This representation is called the *display form.* A *linear form* of representation is shown in Fig. 3.2.

```
[ UART ] -
        ( agnt )->      [ communicates ] -
                                ( obj )  ->      [ devices ]
        ( loc )  ->     [ chip ]
```

Fig. 3.2 The linear form of a conceptual graph

The words in rectangular braces are concept referents and those in curved braces are conceptual relation referents. This graph can be written differently by choosing another head node as shown in Fig.3.3.

```
[ communicate ] -
        ( agnt )->      [ UART ] -
                                ( loc )  -> [ chip ]
        ( obj )  ->     [ devices ]
```

Fig. 3.3 Equivalent linear form of Fig. 3.2

Hence by changing the direction of arcs between two concepts, the conceptual relation name changes.

The concepts and conceptual relations used to build the conceptual graph depend upon the information to be represented. Thus an explicit graph can be big and unwieldy because of the various concepts and relations being used. Further, various users may interpret information with a different perception and use different names for concepts and conceptual relations as shown in Fig. 3.3. This problem is solved by using a *type label* along with the concept referent. This limits the concept labels and relation labels of the type of knowledge being represented. Hence for each application it is necessary to develop a hierarchy of concept types. Also a list of valid relations is used to limit the type of relations. Using type labels the graph in Fig. 3.2 can be represented by another linear form,

```
[ 1: device : UART ]
        ->      ( act )  ->     [ 2 ]
        ->      ( loc )  ->     [ 4 ]
[ 2 : action : communicate ] -
        ->      ( obj )  ->     [ 3 ]
[ 3 : object : devices ]
[ 4 : object : chip ]
```

Fig. 3.4 Linear form with concept types

In the representation in Fig. 3.4 the notation used for the concept is,

[cid : type : referent]

where,

cid - concept identification number

type - concept type

referent - concept name or label.

A concept can be classified as a *generic concept* if it has a type but no name, e.g.,

[1 : event : *]

[2 : u : null]

or it can be an *individual* if both, a concept type and an identifier are specified, e.g.,

[1 : device : UART]

[2 : id : EN]

In the above notation a * and null are equivalent.

3.3 Conceptual Graph Operations

There are various terms related to conceptual graphs and operations performed on them. To understand all of those used in the CGVHDL Linker, a brief discussion of them follows.

3.3.1 Canonical Graphs

As mentioned in the earlier section, a conceptual graph is a combination of concepts and conceptual relations connected through directed arcs. However, not all conceptual graphs may make sense as shown in Fig. 3.5, thus some graphs are defined to be *canonical.* The graph in Fig. 3.5 can be interpreted as the sentence, "an event rise in the location REG has a size device", which conveys no valid idea.

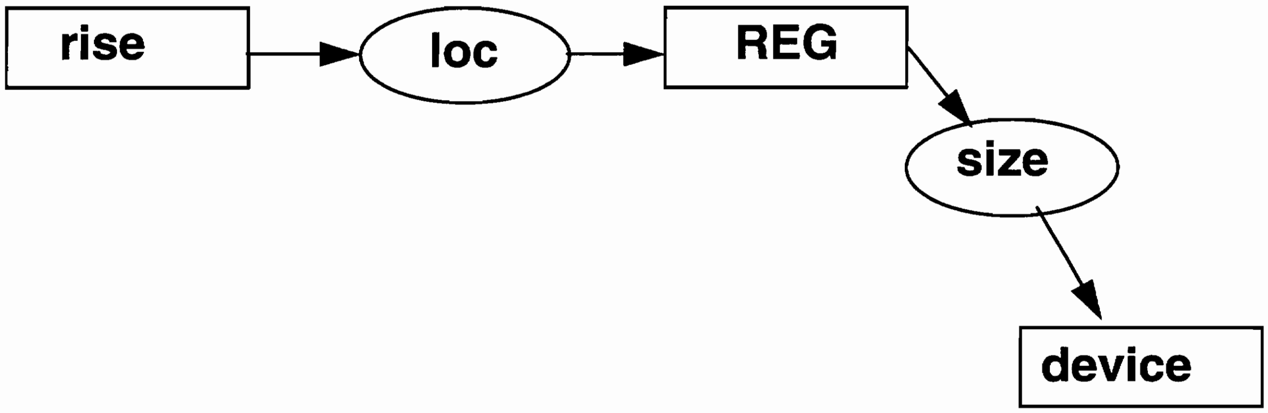


Fig. 3.5 A graph that does not make sense

Definition.

*A **canonical graph** represents a valid idea or a perception of a true situation. [Sowa 84]*

Thus all canonical graphs can be used to represent information that is meaningful and so the concepts and relations in the graph are chosen according to semantic rules and constraints. A graph is *canonized* by,

i. the graph assembler during perception.
ii. deriving it from other canonical graphs by using formation rules.
iii. arbitrarily declaring it to be canonical.

All graphs used within the scope of this thesis are canonical. The most common way of obtaining canonical graphs is by deriving them which then requires a starting set of basic graphs. These basic graphs are called a *canonical basis.*

Definition.

*A **Canonical Basis** B = { B_1 , B_2 , B_3 , B_4 . . . } is a finite set of well formed conceptual graphs.*

Fig. 3.6 shows a graph which may be a part of a canonical basis. It is a graph for the action write and shows all the outgoing relations that may be possible. All referents shown in the graph are generic. A canonical basis contains many such graphs that can be used to put together a larger graph to represent some semantic or logical information.

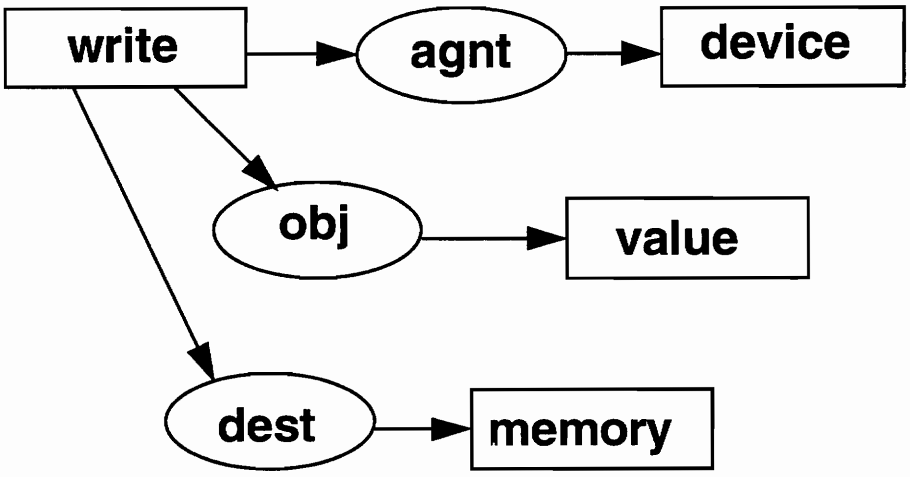


Fig. 3.6 A canonical graph

The Semantic Analyzer in the ASPIN system uses a *canon* to generate its conceptual graph output.

Definition.

*A **canon** contains the necessary information to derive a canonical graph, i.e.,*

i. a type hierarchy T

ii. a set of individual referents I

iii. a conformity relation :: that relates labels in T to referents in I.

iv. a finite set of conceptual graphs, i.e., a canonical basis.

[Mugnier, Chein 93]

Examples of a type hierarchy and a set of referents for conceptual relations are listed in Appendix D.

The canonical formation rules used to derive a canonical graph G' from G_1 and G_2 are,

i. *Copy (c)* : G' is a copy of G.

ii. *Restrict (r)* : Any concept c in G is replaced by a subtype.

iii. *Join (j)* : If a concept c_1 is identical to a concept c_2, then delete c_2 and connect to c_1 all arcs from conceptual relations that were connected to c_2. If c_1 and c_2 belong to different graphs then it is an external join j_e, whereas if c_1 and c_2 belong to the same graph then it is an internal join j_i.

iv. *Simplify (s)* : If concepts c_1 and c_2 are identical in a graph then either one can be deleted along with all the connecting arcs. *[Sowa 84]*

3.3.2 Covering a Canonical Graph

Definition.

A cover of a graph G, is a set of connected subgraphs of G such that every vertex and every edge of G appears in at least one of these subgraphs.

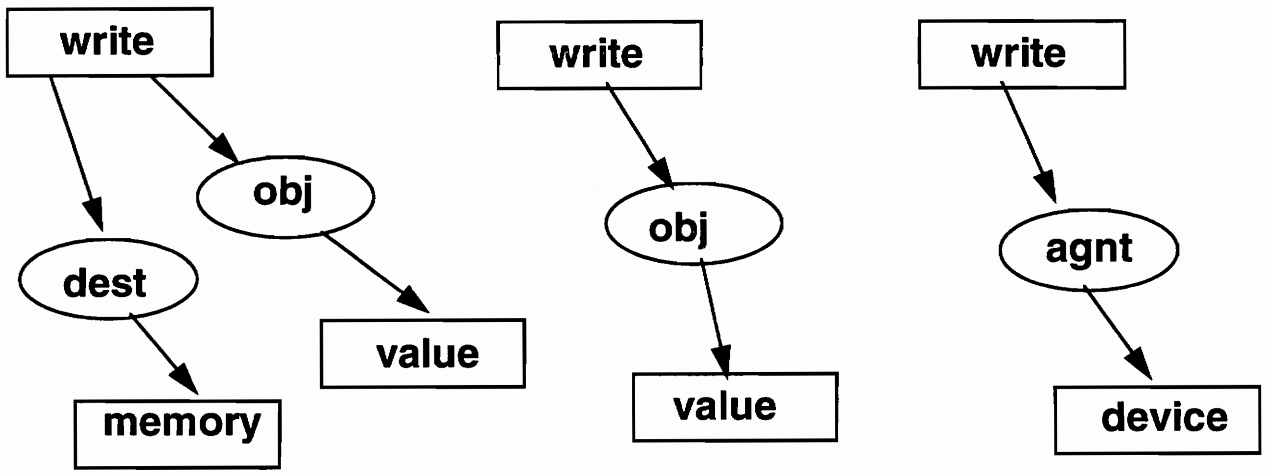


Fig 3.7 Cover of the conceptual graph in Fig. 3.6

A cover of a canonical graph can be derived from the original graph by using a set of internal operations on the graph, i.e. *r, s,* j_i. It can be seen that all the relation vertices have to be covered in order to cover the whole graph. The process of generating a cover for a graph G is called *covering*. Fig. 3.7 shows a cover for the conceptual graph shown in Fig. 3.6.

3.4 Conceptual Graphs in ASPIN

The ASPIN system has various components and each of them generate conceptual graphs that are integrated by the conceptual graph processor. The CGVHDL linker can currently process conceptual graphs that were generated from sentences using the ASPIN system's parser and semantic analyzer. The semantic analyzer uses a canon which generates a conceptual graph such that the head is an action or behavior concept, e.g., write, rise etc.

A simple example of a shift-register is shown in Fig.3.8. The sentences used to describe it are,

i. *The 8 bit data is loaded into the SREG register when LD rises.*

ii. *The data in SREG is shifted left onto SOUT when SL is high and CLK rises.*

iii. *The data in SREG is output to OP.*

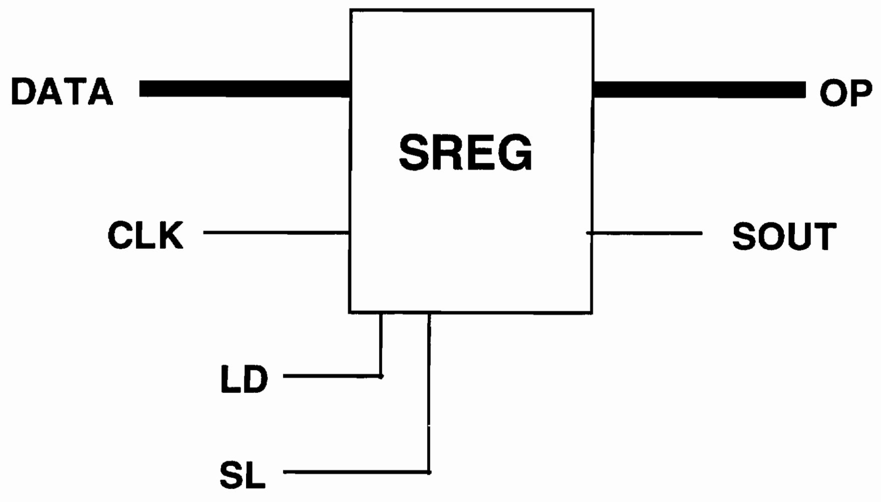


Fig. 3.8 A Parallel Load Shift Register

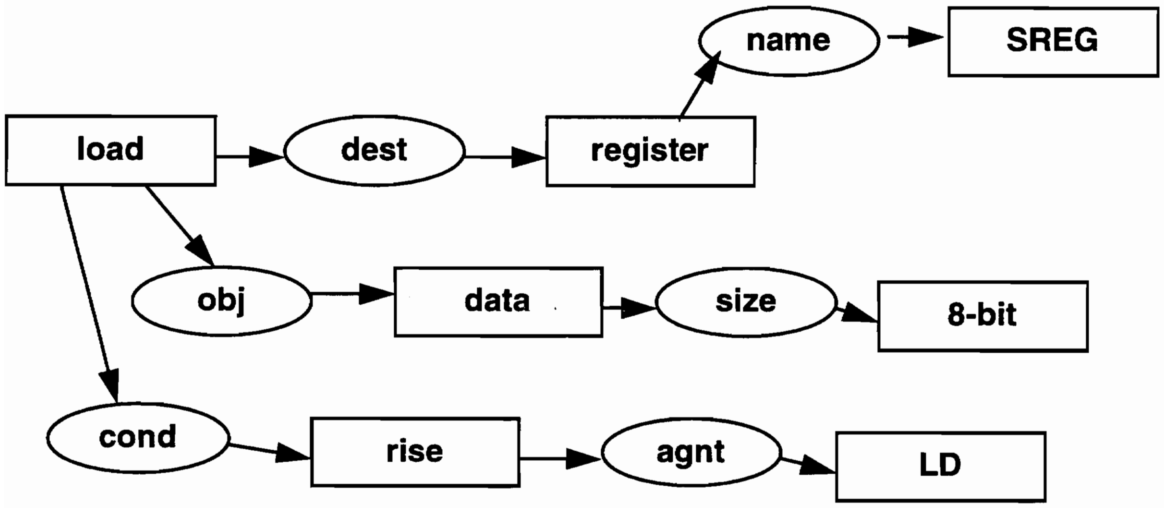


Fig. 3.9 The conceptual graph for sentence (i)

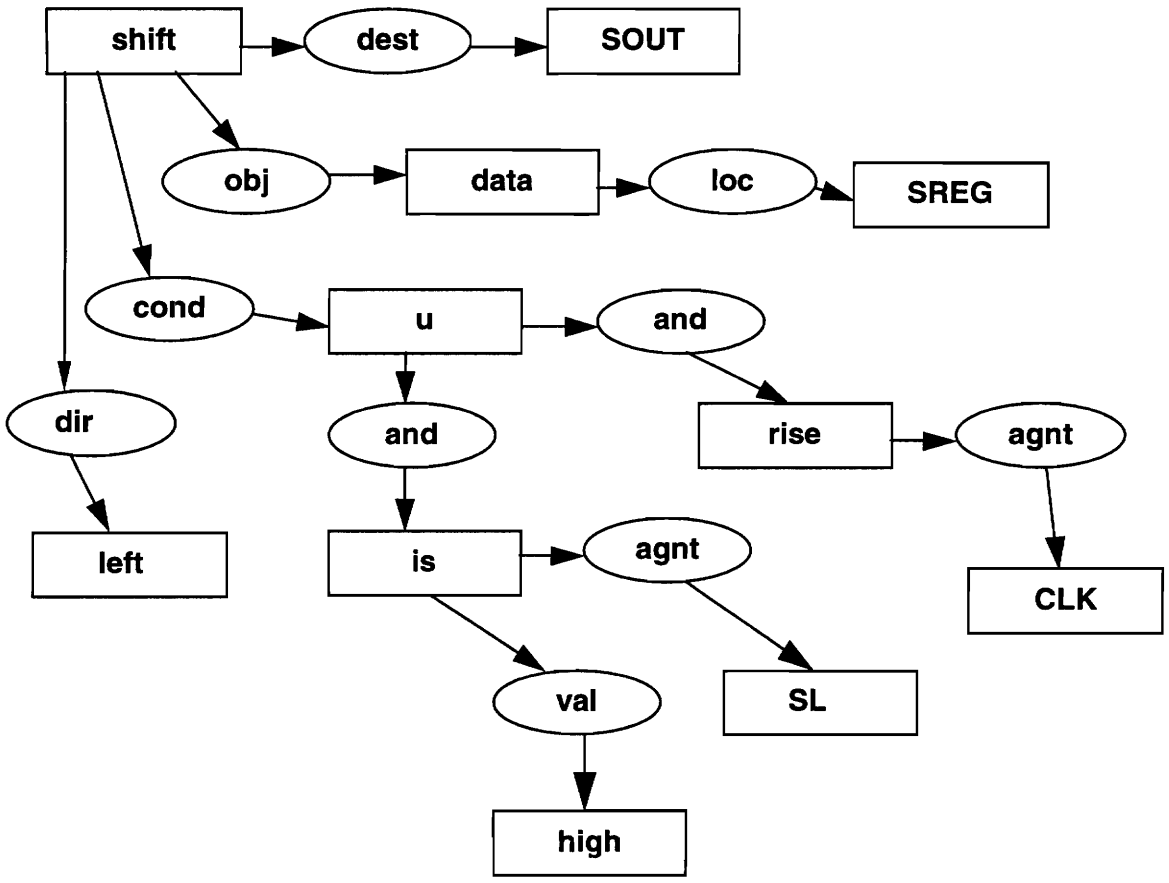


Fig 3.10 Conceptual graph for sentence (ii)

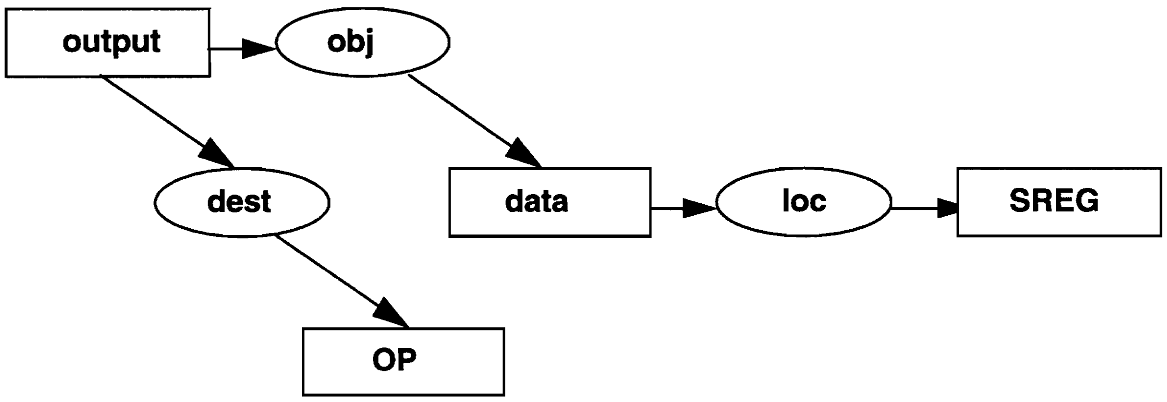


Fig. 3.11 Conceptual graph for sentence (iii)

The three graphs shown in Fig.3.10 - 3.12 are merged by the conceptual graph processor to form a single conceptual graph that forms the input to the CGVHDL linker. The conceptual graph processor uses the output of a coreference detector to identify common concepts and then uses a '*same as*' relation or a join to integrate the graphs. Various examples of complete graphs are in Appendix A.

Chapter 4

The Projection Operator

4.1 Introduction

The first version of the CGVHDL Linker used to identify action concepts, object concepts and condition - activity links to generate VHDL constructs. This approach led to the generation of simple VHDL code; i.e., each process contained one behavior such as a reset event, a shift or a load event described by one VHDL construct. The new algorithm for the CGVHDL Linker takes a different approach to its task of transforming a conceptual model of a system into a behavioral VHDL model. The MODAS application contains a library of tested VHDL primitive processes commonly used in digital design. It was proposed that, instead of generating VHDL code by constructs, the CGVHDL Linker performance would be improved if a MODAS primitive process was recognized from a conceptual graph, and the corresponding VHDL code be read from a respective file and then modified according to the input specification. The new approach uses an algorithm that parses an integrated conceptual graph. It uses a type hierarchy along with a referent database to identify concepts that imply certain primitives from the MODAS primitive library. Upon identification, it looks up a *schema* of the identified process and tries to instantiate a prototype from the input conceptual graph and the VHDL code from the MODAS primitive library.

In designing the new algorithm, the various system level aspects considered were,

i. a way of storing a database of *schemata* (file system organization).

ii. a representation scheme for the schemata (a conceptual graph, a set of canonical graphs or a cover).

iii. a referent and type dictionary and a database to look up concept types, for identifying them as primitive related concepts.

iv. and lastly, but the most important, a scheme for graph matching between the primitive schema and the input conceptual graph (graph matching and projection).

In this chapter we will address the aspect of graph matching.

4.2 Schemata : Generalization and Specialization

The conceptual model used in this application is a *conceptual graph.* A conceptual graph consists of concepts connected to each other through a relation arc. Each primitive process in the Modeler's Assistant library has a certain behavior exhibited by it and its conceptual model will have concepts related to each other with conceptual relations. Such a conceptual model is referred to as a *schema*. A schema consists of a network of concepts which *stereotype* an entity. A schema is defined formally as follows,

Definition.

The basic structure for representing background knowledge for human like inference is called a schema. *[Sowa 84]*

The schema of a primitive is a conceptual graph of the primitive entity, e.g., a counter, which describes all the behavior using action and behavior concepts. The VHDL code for a counter shown in Fig. 4.1 has the associated schema shown in Fig. 4.2. The form of schemata used are basically conceptual graphs which contain generic and individual concepts and conceptual relations to represent all behaviors and activities. Apart from this information it provides the concept numbers for action and behavior concepts and pairs of common events on a signal.

```
if (RESET = 'RESTTRIG' and RESET'EVENT and SYNC /= 'SYNCTRIG') or
        (RESET = 'RESTTRIG' and SYNC = 'SYNCTRIG' and CLK = 'CLKTRIG' and
        CLK'EVENT) then
                CNT <= "#CNT.zeros" after CNT_DEL;
elsif (LOAD = 'LOADTRIG' and CLK = 'CLKTRIG' and CLK'EVENT) then
                CNT <= DATA after CNT_DEL;
elsif (EN = 'ENTRIG' and CLK = 'CLKTRIG' and CLK'EVENT) then
                if UP = 'UPTRIG' then
                        CNT <= INC(CNT) after CNT_DEL;
                else
                        CNT <= DEC(CNT) after CNT_DEL;
                end if;
end if;
```

Fig 4.1 VHDL Description for a counter primitive

```
[  1 : device : counter ]
            ->( name : null )  -> [ 2 ]
[  2 : id : CNT ]
[  3 : event : reset ]
            ->( obj : null )  -> [ 1 ]
            ->( attr : null )  -> [ 6 ]
            ->( cond : when )  -> [ 13 ]
[  4 : event : increment ]
            ->( obj : null )  -> [ 1 ]
            ->( cond : when )  -> [ 22 ]
[  5 : state : is ]
            ->( agnt : null )  -> [ 1 ]
            ->( attr : null )  -> [ 12 ]
[  6 : constant : 0 ]
[  7 : event : rise ]
            ->( agnt : null )  -> [ 8 ]
[  8 : id : RESET ]
[  9 : event : rise ]
            ->( agnt : null )  -> [ 10 ]
[ 10 : device : clock ]
            ->( name : null )  -> [ 11 ]
[ 11 : id : CLK ]
[ 12 : constant : 8 ]
[ 13 : u : or ]
            ->( or : null ) -> [ 14 ]
            ->( or : null ) -> [ 19 ]
[ 14 : u : and ]
            ->( and : null ) -> [ 7 ]
            ->( and : null ) -> [ 15 ]
[ 15 : state : is not ]

            ->( attr : null ) -> [ 17 ]
[ 16 : id : SYNC ]
[ 17 : constant : 1 ]
[ 18 : state : is ]
            ->( agnt : null ) -> [ 16 ]
            ->( attr : null ) -> [ 6 ]
[ 19 : u : and ]
            ->( and : null ) -> [ 20 ]
            ->( and : null ) -> [ 21 ]
            ->( and : null ) -> [ 9 ]
[ 20 : state : is ]
            ->( agnt : null )  -> [ 8 ]
            ->( attr : null ) -> [ 17 ]
[ 21 : state : is ]
            ->( agnt : null )  -> [ 16 ]
            ->( attr : null ) -> [ 17 ]
[ 22 : u : and ]
            ->( and : null ) -> [ 9 ]
            ->( and : null ) -> [ 23 ]
            ->( and : null ) -> [ 30 ]
```

```
[ 23 : state : is ]
            ->( agnt : null ) -> [ 24 ]
            ->( attr : null ) -> [ 17 ]
[ 24 : id : EN ]
[ 25 : event : decrement ]
            ->( obj : null )  -> [ 1 ]
            ->( cond : when )  -> [ 26 ]
[ 26 : u : and ]
            ->( and : null ) -> [ 9 ]
            ->( and : null ) -> [ 27 ]
            ->( and : null ) -> [ 28 ]
[ 27 : state : is ]
            ->( agnt : null ) -> [ 24 ]
            ->( attr : null ) -> [ 17 ]
[ 28 : state : is ]
            ->( agnt : null ) -> [ 29 ]
            ->( attr : null ) -> [ 6 ]
[ 29 : id : UP ]
[ 30 : state : is ]
            ->( agnt : null ) -> [ 29 ]
            ->( attr : null ) -> [ 17 ]
#1,3,4(15:18)(18:15)
```

Fig 4.2 Schema of the COUNTER primitive

The numbers after the hash specify the main behavior and action concepts of the schema. The pair 15 and 18 specify two different states for the signal SYNC, i.e., 0 or 1. This helps to find differences between the input conceptual graph and the schema during projection.

Definition.

If a ***conceptual graph*** *u is canonically derivable from a conceptual graph v (possibly with the join of other conceptual graphs w_1, w_2...,w_n), then u is called a specialization of v, written u<= v, and v is called a generalization of u. [Sowa 84]*

The definition implies that a graph can be a generalization of itself. A graph can be generalized by replacing the type of an original concept with a super type, by deleting relations, or by deleting concepts and all attached relations. The graph shown in Fig. 4.3 is a generalization of the counter schema shown in Fig. 4.2.

```
[  1 : device : counter ]
        ->( name : null )  -> [ 2 ]
[  2 : id : OCTR ]
[  3 : event : reset ]
        ->( obj : null )  -> [ 1 ]
        ->( attr : null )  -> [ 6 ]
        ->( cond : when )  -> [ 7 ]
[  4 : event : increment ]
        ->( obj : null )  -> [ 1 ]
        ->( cond : when )  -> [ 9 ]
[  5 : state : is ]
        ->( agnt : null )  -> [ 1 ]
        ->( attr : null )  -> [ 12 ]
[  6 : constant : 0 ]
[  7 : event : rise ]
        ->( obj : null )  -> [ 8 ]
[  8 : id : OCLKEN ]
[  9 : event : rise ]
        ->( agnt : null )  -> [ 10 ]
[ 10 : device : clock ]
        ->( name : null )  -> [ 11 ]
[ 11 : id : OCLK ]
[ 12 : constant : 8 ]
```

Fig 4.3 Generalization of the counter schema

4.3 Projection

If a graph G' exists such that it is a specialization of G, then there is a subgraph S that can be mapped from the graph G' to G. S represents G in some way (topology and information represented), and additional graphs can be joined to it for a canonical derivation of G'. The subgraph S is called a *projection* of G in G'. Projection of graphs, denoted by the greek letter Π , can be written as,

$$S = \Pi(G)$$

Every relation in the conceptual graph S must be identical to the corresponding relation in G. The concepts in S must be of the same type or subtype. A formal definition of projection is as follows,

Definition.

*Given two conceptual graphs G and G', a **projection** Π from G to G' is an ordered pair of mappings from (C_G, R_G) to $(C_{G'}, R_{G'})$, such that :*

(i.) For all edges rc of G, $\Pi(r)\Pi(c)$ *is an edge of G'.*

(ii.) $\forall\ r \in\ R_G$, $type(\Pi(r)) = type(r)$.

(iii) $\forall\ c \in\ C_G$, $type(\Pi(c)) <= type(c)$

If c is individual, then

$ref(\Pi(c)) = ref(c)$

[Mugnier, Chein 92]

4.4 Graph Matching

A graph matching operator is a second class of operations used to find a correspondence between two graphs (e.g., the input specification graph and the schema). Two graphs G_1 and G_2, are said to match if there is a correspondence of a certain type between two *maximal* subgraphs of G_1 and G_2, say G_1' and G_2' . Some of the parameters used to specify graph matching are:

(i) Compatibility criterion between the c-vertex labels: Typically the graphs are matched to obtain a common specialization. Thus two concepts are compatible if they have some common subtypes; the original concepts are super types of the concept in the specialization. If G_1 and G_2 are to be mapped, then the label of any c-vertex in G_1' may be greater than or equal to its image.

(ii) The type of correspondence between vertex sets of G_1' and G_2': The correspondence may be surjective (each concept in G_2 can be matched with a concept in G_1), or bijective (each concept in G_2 can be matched with a unique concept in G_1), or a mapping.

(iii) The definition of *maximality*: Three definitions are possible. There is a *cardinal maximality* if the cardinal of G_1' and/or G_2' is maximal. If no subgraphs of G_1 and/or G_2 extend G_1' and G_2', then there is an *inclusion maximality*. Finally, if the correspondence between G_1' and G_2' cannot be extended then there is a *correspondence inclusion maximality*.

[Mugnier, Chein 92]

4.5 Mugnier - Chein Projection

This is the scheme implemented in the CGVHDL Linker. Consider as an example the projection of a tree on a graph, that is usually the case as the input from the Conceptual Graph Processor will have formed out of external joins with other graphs. Let the tree be denoted as T and the graph as G as shown in Fig. 4.4. Let the node 'a' in T be the head node for starting the projection operation.

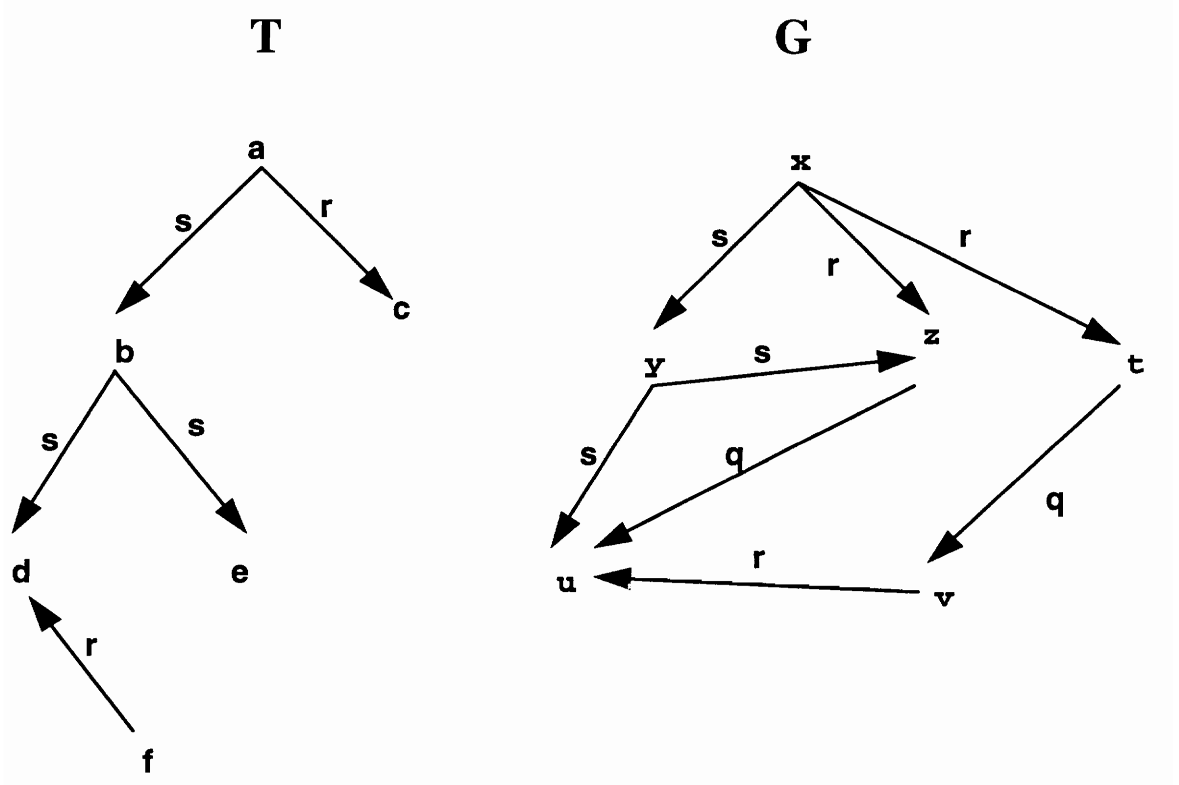
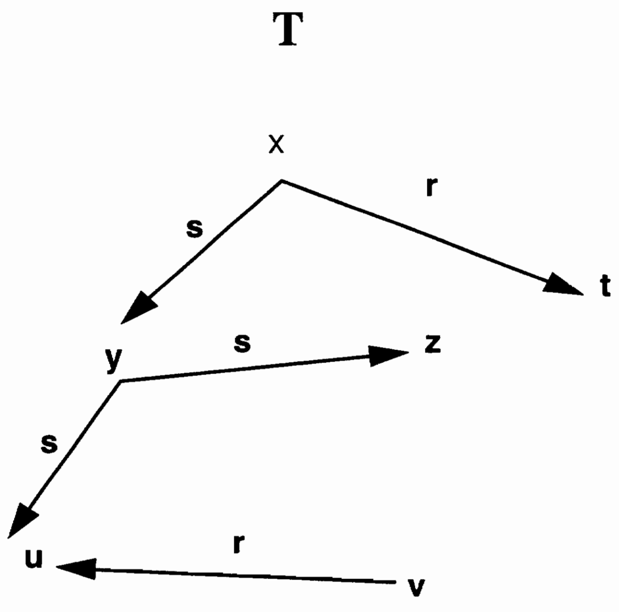


Fig 4.4 A Tree and a Graph

From figure 4.4, applying the projection definition could give us two possible results shown in figure 4.5. The result of a projection can be *injective* (one to one) if Π (G) in G' is very similar to G. To obtain an injective projection, each element (concept or relation) of Π(G) must have a unique corresponding element in G. The injective projection of a tree to a tree is computable in polynomial time but that of a tree to a graph is NP complete. Hence in the general case (i.e., a graph with concepts and relations) an injective projection is made locally injective, i.e., each subtree of the successors of a node in G is projected to the successors of the matching node in G'. The projection is locally c-injective if it is injective on the successor set of each r-vertex. It may also be locally r-injective if the projection is injective on the successor set of each c-vertex.



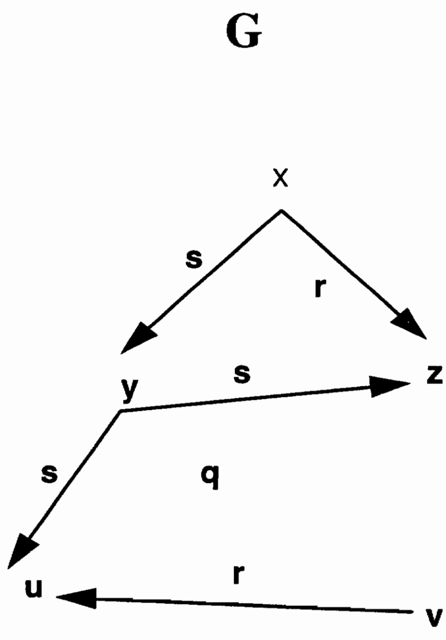


Fig 4.5 (a) Injective projection **Fig 4.5(b) Locally c-injective projection**

In the general case of projection, relation vertices play an important role. A successor is a concept vertex along the direction of a relational edge from the head concept. The successors of a head concept vertex are the relation vertices. The successors of the relation vertices are the neighbor concept vertices or the sons of the head concept.

The general case of projection can now be defined and recursively applied as,

Π *is a projection from T to G, with* Π *(a) = c, if it satisfies,*

I. *(i) ref(a) = ref(c)*

(ii) For all r successor of a, r' = Π *(r) is a neighbor c such that*

(ii-1) type (r) = type(r')

(ii-2) $P_r[a] \subseteq P_{r'}[c]$*, and,* *[Mugnier & Chein 92]*

II. *(ii - 3) for all a_i successor of r, v = Π (a_i) is the neighbor of r'*

such that,

(ii-3-1) $P_r[a_i] \subseteq P_{r'}[v]$,

(ii-3-2) Π $_{Ti}$, the restriction of Π to the subtree T_i

induced by a_i, is a PROJECTION from T_i to G.

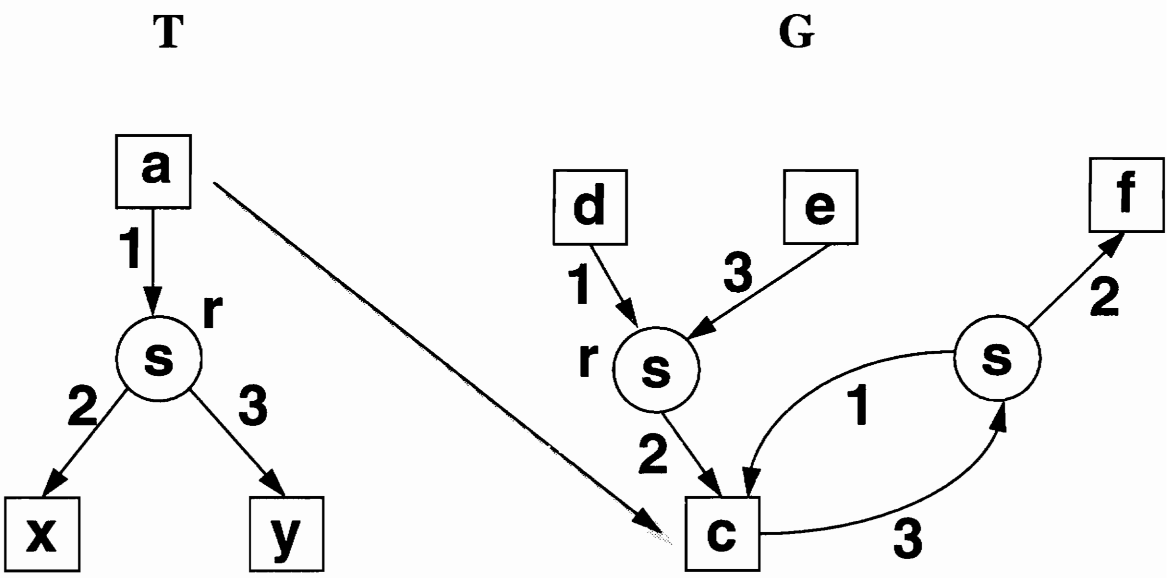


Fig. 4.6 Projection in the general case

This algorithm causes a depth first search of relations and concepts. This algorithm is easily programmable using two crossed recursive functions.

In Fig. 4.6, the graph G is searched for a node c such that type(a) >= type(c), ref(a) = ref(c) or , i.e., Π(a) = c. The projection give the results,

c = Π (a) = Π (y)

r_2 = Π (r)

f = Π (x)

4.6 Matching of graphs using a cover of a tree

Another approach to matching graphs involves using the cover of a conceptual graph. For example take the example of a tree to be matched with a schema as shown in Fig. 4.4. Instead of storing the conceptual graph of the schema , the database contains a cover of the schemata conceptual graph.

$$\begin{tikzcd} a \arrow[r] & b \\ a \arrow[r] & c \\ b \arrow[r] & d \\ b \arrow[r] & e \\ d \arrow[r] & f \end{tikzcd}$$

$$\begin{tikzcd} & b \\ a \arrow[ur] \arrow[dr] & \\ & c \end{tikzcd}$$

$$\begin{tikzcd} & e & \\ b \arrow[ur] \arrow[dr] & & \\ & d \arrow[r] & f \end{tikzcd}$$

Fig. 4.7 Two ways of representing the cover of the tree in Fig. 4.4

The left column shows a cover represented as *star graphs* (each graph has two concepts and one relation). The next step of the covering process involves taking each graph of the cover set and projecting it over the input graph G as shown in Fig. 4.8. The right column shows subgraphs of the tree in Fig. 4.4 which also form a cover. The first algorithm such subgraphs for mapping the schema to the conceptual graph. This method was dropped due to the complexity involved in making subgraphs and maintaining them if the schema were to be changed.

[Mugnier, Chein 92]

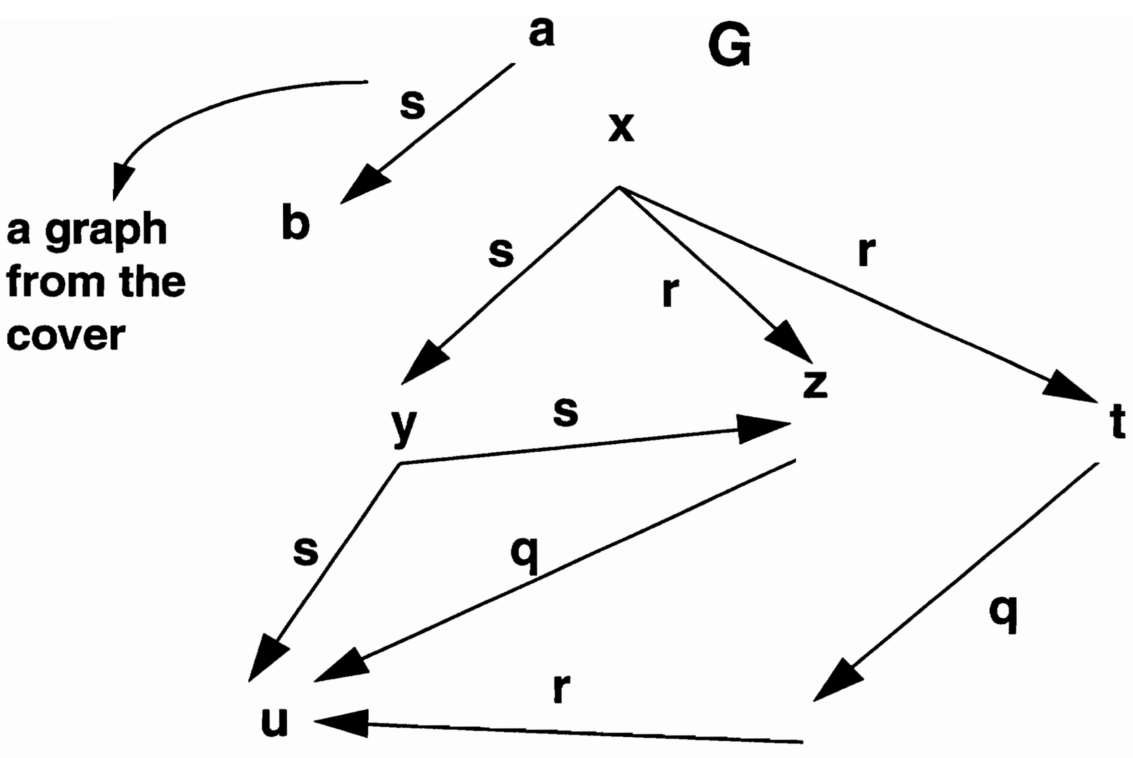


Fig. 4.8 Covering a graph by placing a subgraph of over G

After using all the graphs in the cover set, the projection results are combined to evaluate the matching of T to G. A graph matching scheme of this type has been implemented [Yang,Chui 92] for designing a database query protocol. This scheme is widely used in the design of object oriented database systems.

4.7 Graph Matching using Myaeng - Lopez Algorithm

The Myaeng - Lopez graph matching algorithm is a different approach than the previous two techniques. This algorithm was found to be very complex and is not used by the CGVHDL Linker. Since it is computationally intensive it is split in two halves to alleviate that problem. The first half of steps deal with finding the maximal common subgraphs for two graphs. These steps ignore all the information in the conceptual graph like labels and application specific data and treat it like a directed bipartite graph with no labels. The second half then processes all the semantic and structural information. The algorithm can be made more flexible by adding some application heuristics.

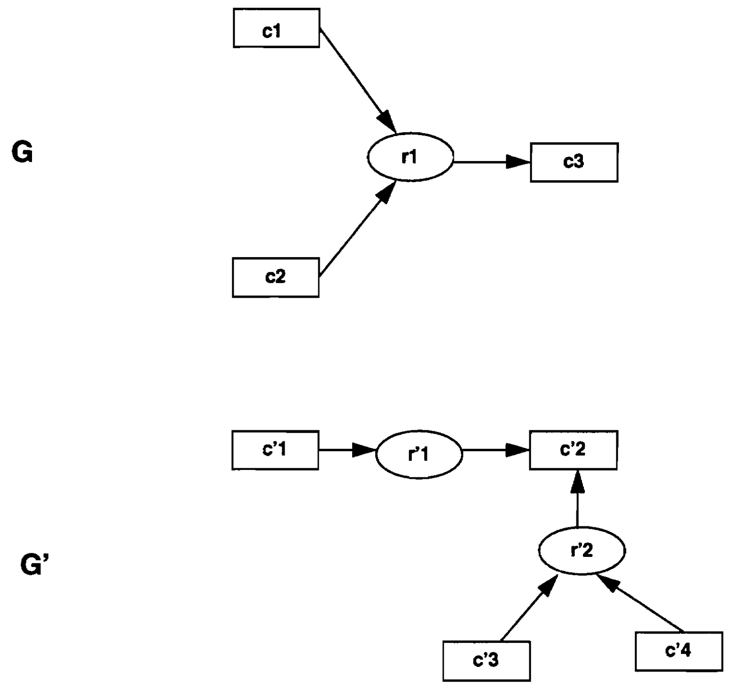


Fig 4.9 Graphs to be matched

Consider the two graphs shown in Fig. 4.9. Using the notation as shown below,

$G = (C,R,E)$ and $G' = (C', R', E')$

where,

$C = \{ c_1, c_2, c_3\}$, $R = \{ r_1\}$, $E = \{(c_1, r_1), (c_2, r_1), (r_1,c_3)\}$

and,

$C' = \{ c'_1, c'_2, c'_3, c'_4 \}$, $R = \{r'_1, r'_2\}$,

$E' = \{(c'_1,r'_1), (c'_3,r'_2), (c'_4,r'_2), (r'_1,c'_2), (r'_2,c'_2)\}$

An intermediate structure called an association graph is generated such that,

$G_A = (C_A, R_A ,E_A)$

where

$C_A = C * C'$

$R_A = R * R'$

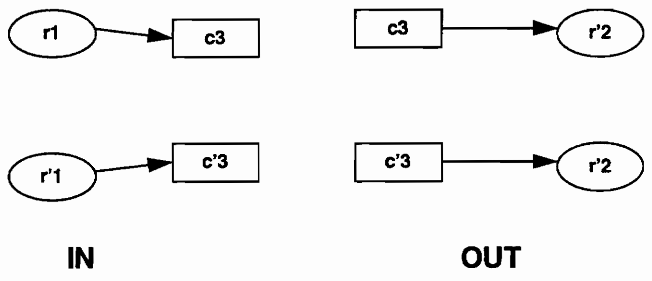
E_A is defined as,

$((c_i , c'_j) , (r_k , r'_l)) \in E_A$ if $(c_i , r_k) \in E$ and $(c'_j , r'_l) \in E'$

$((r_i , r'_j) , (c_k , c'_l)) \in E_A$ if $(r_i , c_k) \in E$ and $(r'_j , c'_l) \in E'$

The G_A is then used to identify the structural information of the two graphs. Now for each node in G and G', adjacent nodes are found with the same directionality. This is done by taking each node in G_A and finding out all the incoming nodes and outgoing nodes. This step establishes local connection information used later to construct the maximal subgraph. These local connections are then combined by selecting a node and it's IN and OUT set of nodes.

Continuing this process a final set of maximal subgraphs can be obtained. The two sets are thus combined and the redundant sets are eliminated. For the example in Fig. 4.9, the matched Maximal subgraphs are shown in Fig. 4.10. This basic algorithm is now extended by heuristics. For example to match a graph with labels, we add constraints in the algorithm during the formation of the associative graph to block concepts with different labels from being associated. This algorithm as seen is very complex as each graph is split into parts of two elements i.e. a relation and a concept directionally arranged. The associative graph has a large number of entries and gets very difficult to handle with a large graph. Though many heuristics can be used to alleviate the complexity, the algorithm is being analyzed by *[Myaeng-Lopez 91]* for complexity and worst case performance.



The associatice set generated lists

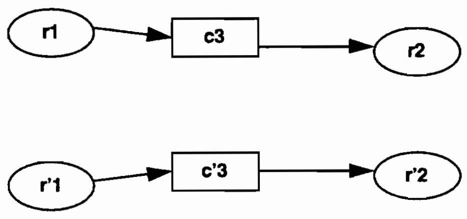


Fig 4.10 An example of joins during formation of maximal matched subgraphs

Chapter 5

CGVHDL Linker: The Projection Based Algorithm

5.1 Introduction

The previous chapter explained the basic definitions and groundwork involved in deciding the final algorithm for projection. Of the three schemes explained in the previous chapter it was decided to go ahead with the Mugnier-Chein projection Algorithm. This decision was taken because,

(i) The Mugnier-Chein tree to graph algorithm could be implemented easily using recursive functions and the design of the object oriented data structures favored this implementation.

(ii) Secondly, the projection operation needed a library of schema's to be used during the projection operation. The library of schema's is comprised of conceptual graphs constructed to generalize the behavior of various primitive processes in the MODAS primitive library. Since a CAD system requires easy system library maintenance, after some deliberation it was decided that a schema is easier to maintain and develop than a set of graphs forming a cover for a generalized entity's conceptual graph.

Apart from the library of schema, the projection algorithm also needs a type hierarchy to apply the rules of projection to the input conceptual graph. This involves the use of the concept hierarchy and relation database which was developed for the first version of the linker *[Honcharik 93]*. Once the conceptual graph is read in all its concepts and relation are classified using these databases.

5.2 Processing Units in the CGVHDL Linker

The CGVHDL has the following sub functional blocks as shown in Fig. 5.1,

(i) The Input Preprocessor: This has the task of setting up the taxonomy data structure. It also reads in the input conceptual graph and stores it in a datatype called graphtype.

(ii) The Primitive Detector: Here the graph is parsed to check for concepts which can be related to a primitive in the MODAS primitive library. It generates a table to keep track of these concepts and their schemata.

(iii) The Projection Engine: Each of the entries in the table made by the Primitive Detector is used to perform a projection on the input conceptual graph to identify the reference to a primitive. The input conceptual graph and a schema for a primitive

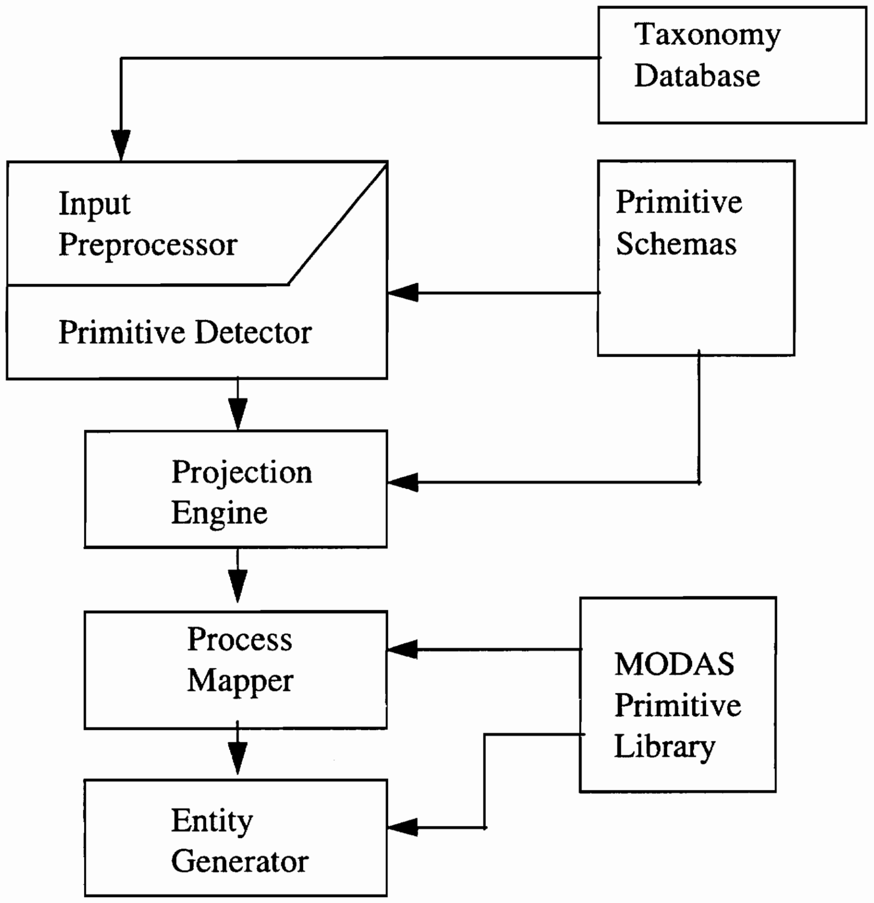


Fig. 5.1 Block Diagram of the CGVHDL Linker

are traversed and simultaneously a projection operation is performed between them. Every projection call generates a subgraph of the input graph which refers to a primitive. After all projections are complete, all subgraphs of the input are searched for common concepts and a list of them is passed on to the Entity Generator.

(iv) Process Mapper: This unit uses the subgraph to instantiate a process for the process model graph of the input. It modifies the referents of signals, generics and variables to new labels as in the specification. It also prompts the user for referents not provided in the specification.

(v) Entity Generator: This stage uses the list of common concepts generated by the Projection Engine to generate signal interconnections between the processes. Lastly, the user is prompted for final changes before writing the MODAS files for the entity.

5.3 The Algorithm for the CGVHDL Linker

The steps performed by the CGVHDL Linker for it's task of transforming a conceptual graph to VHDL are enumerated below. These steps are performed after the input conceptual graph has been read and the taxonomy and the relation databases have been setup.

(i) Identify Concepts in the input Conceptual Graph G which refer to an instance of a primitive in MODAS.

The input conceptual graph is searched for concepts which may refer to primitives in the MODAS primitive library. For example consider a section of the input graph shown in Fig. 5.2,

```
[ 1 : write : load ]                    /* may be referring to a form of
        ->( gindx : 1 )                 register */
        ->( obj : null ) -> [ 2 ]
        ->( dest : into ) -> [ 4 ]
        ->( cond : when ) -> [ 5 ]
[ 2 : data : data ]
```

```
        ->( name : null ) -> [ 3 ]
[ 3 : id : DATA ]
[ 4 : device : shift-register ]              /* definitely refers to a
        ->( name : null ) -> [ 6 ]           shift-register */
[ 5 : event : rise ]
        ->( agnt : null ) -> [ 7 ]
[ 6 : id : OREG ]
[ 7 : id : LOAD ]
```

Fig. 5.2 Section of the input conceptual graph

All concepts of type device may have a primitive in MODAS. Further, other device related concepts are identified by looking them up in a primitive index database which links the various possible concept names which may call up a MODAS primitive. The format of such a database file is shown in the Fig. 5.3,

shift-register	SHIFTREG
shifter	SHIFTREG
shift	SHIFTREG
shifted	SHIFTREG
clock	OSC
counter	COUNTER
increment	COUNTER
decrement	COUNTER
load	REGISTER
write	REGISTER
register	REGISTER

Fig. 5.3 The primitive index database

The file organization is quite simple. The first column consists of commonly used words that would relate to a MODAS primitive. The second column contains the MODAS name of the primitive that they may be referring to, i.e., SHIFTREG, OSC, COUNTER etc.. All the MODAS primitive files are stored in its primitive library directory and are referenced with the name of the primitive, e.g.,

SHIFTREG.prm	- the primitive used to instantiate a process
SHIFTREG.prg	- the primitive process schema

Also when a concept is identified, it is assigned a priority number. This number is it's row in the primitive index database. Of the various behaviors identified as primitives, it is possible that two behaviors can be performed by a single primitive, e.g., a counter can load a parallel data word as well as increment or decrement its data. Hence, concepts related to the counter would refer to a counter primitive as well as a register primitive. This would generate instances of processes which are combined in one process and is illegal in VHDL syntax. To solve this problem, more complex primitives are assigned a lower number and are used for projection first. This problem is further addressed in the next steps.

(ii) Each concept identified from (i) is added to a table pointing to the corresponding primitive graph loaded from a library.

Each concept identified from (i) is added to the primitive entry table that has a structure as shown below,

```
entry {
        concept->cid
        concept->used_flag
        filename
        hierarchy
        graph *schema
      }
```

where,

concept->cid: is a concept id number in the input graph.
concept->used_flag: is a flag that is 0 if the concept has not been mapped yet else it is 1.
filename: filename.mod and filename.prm are the process and primitive files in the MODAS primitive library.
hierarchy: is a number indicating the complexity of the primitive
graph : is a pointer to the graph structure storing the schema for projection.

The table is accessed for supplying a head node for beginning every projection operation. This operation can be visualized as shown in Fig. 5.4.

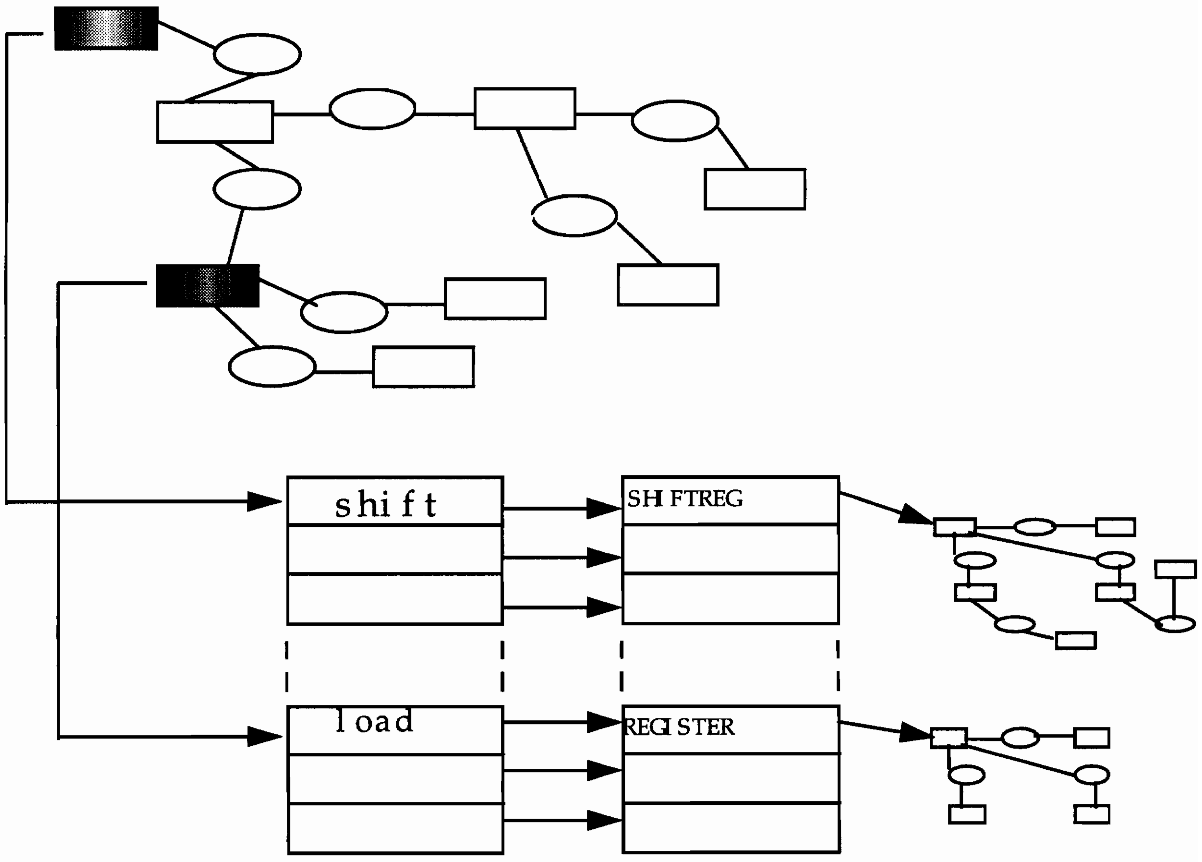


Fig. 5.4 Table structure

The hierarchy entry represents the row in the primitive index database and it specifies the complexity of the primitive. If the entry in the table is a part of another primitive already mapped by a previous projection operation the concept->used_flag will be set to 1. In such a case the entry in the table will not be used to start a projection with it as a head node. The table is sorted so that the concepts with the higher hierarchy field are projected first, this way projection operations can be saved if the table had some entries which got used in a single projection operation.

(iii) **The first entry from the table is made a head concept for the projection of the primitive conceptual graph onto the input graph. This maps behavior and object concepts of the MODAS primitive and marks them used.**

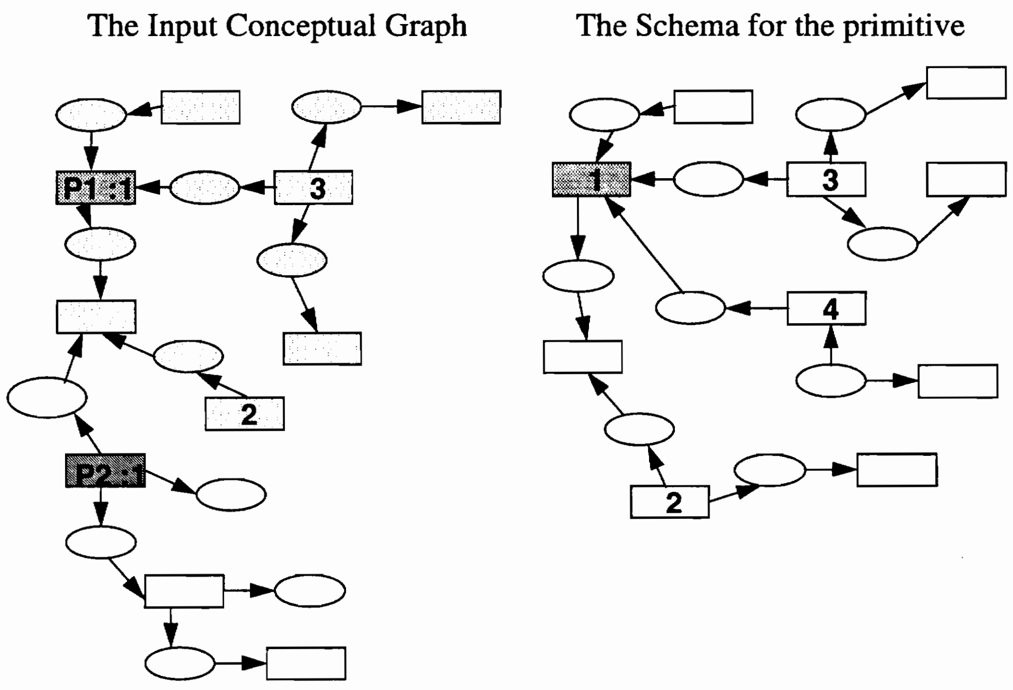


Fig. 5.5 Pre - Projection

As mentioned earlier, the projection algorithm used for graph searching and matching was designed based on the Mugnier-Chein polynomial projection algorithm. Each time the projection function is called, the program performs a depth first graph traversing algorithm using recursive programming techniques. The first entry in the table is probably the most complex primitive. A subgraph of Fig. 5.5 is shown in Fig. 5.6.

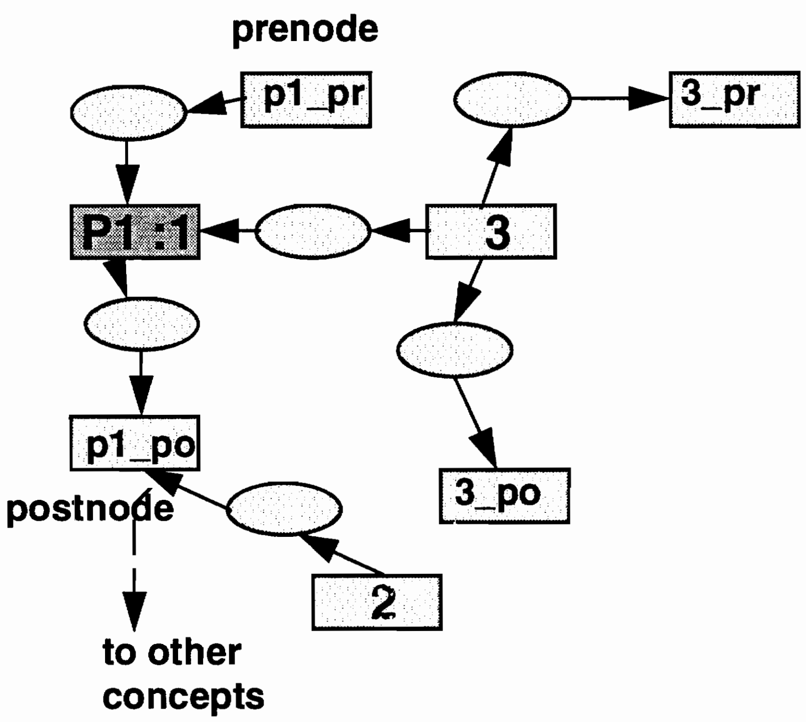


Fig. 5.6 A subgraph of the input conceptual graph

For the graph shown in Fig. 5.5, P1 and P2 are two entries in the primitive entry table. The concepts labeled 1,2 and 3 are behavior concepts in the primitive schema and the input. The concepts p1_pr, 3_pr are prenode concepts of 1 and 3 respectively, whereas 1_po and 3_po are postnode concepts of 1 and 3 respectively. Additionally 2 and 3 are prenodes of p1_po and P1 respectively. The concept id in the table is used to isolate the concept in the input graph G. Using the referent of this concept, the schema conceptual graph (SCG) is searched for a concept with the same referent and type. If,

$$\text{g\_head} \in G1(C_1, C_2, C_3, ...)$$

$$\text{sg\_head} \in S(C_1, C_2, C_3, ...)$$

$$\text{sg\_head} = \Pi(\text{g\_head})$$

Using the sg_head as the head concept all out going relation arcs are taken one at a time, i.e., depth first, to the successive concept till a null relation pointer is reached, e.g., starting at P1, the search reaches p1_po where there are no more outgoing relation arc. A similar traversal of the input is made making use of the projection operator rules from g_head. A list of ordered pairs is made as we go along with this operation. The list contains a mapping of the concepts in the schema into the input graph G. All concepts used up in G are marked as used. The head concept may also have relation arcs pointing to it. To solve this problem the graph data structure had to be made a doubly linked list allowing easy traversal of the conceptual graph. This also meant extending the projection rules so that mapping could be made backwards up the graph. This was also implemented using a recursive function. This ensures a maximal projection of the schema over the input conceptual graph. The figure Fig. 5.7 illustrates the procedure of mapping along arcs with *pre-node* and *post-node* concepts.

As seen from Fig. 5.5, the schema for the primitive pointed to by P1 is a generalization of the subgraph of P1 in the input conceptual graph. Also it is observed that the schema has more behavior concepts than the subgraph for P1 in the input conceptual graph. The nodes 2,3 and 4 are all postnodes and are easily projected by a recursive routine. The node above 1 is a pre-node and a modification in the data structure is required for proper functioning. The figure 5.7 shows each of the concepts in the schema getting projected into the input conceptual graph.

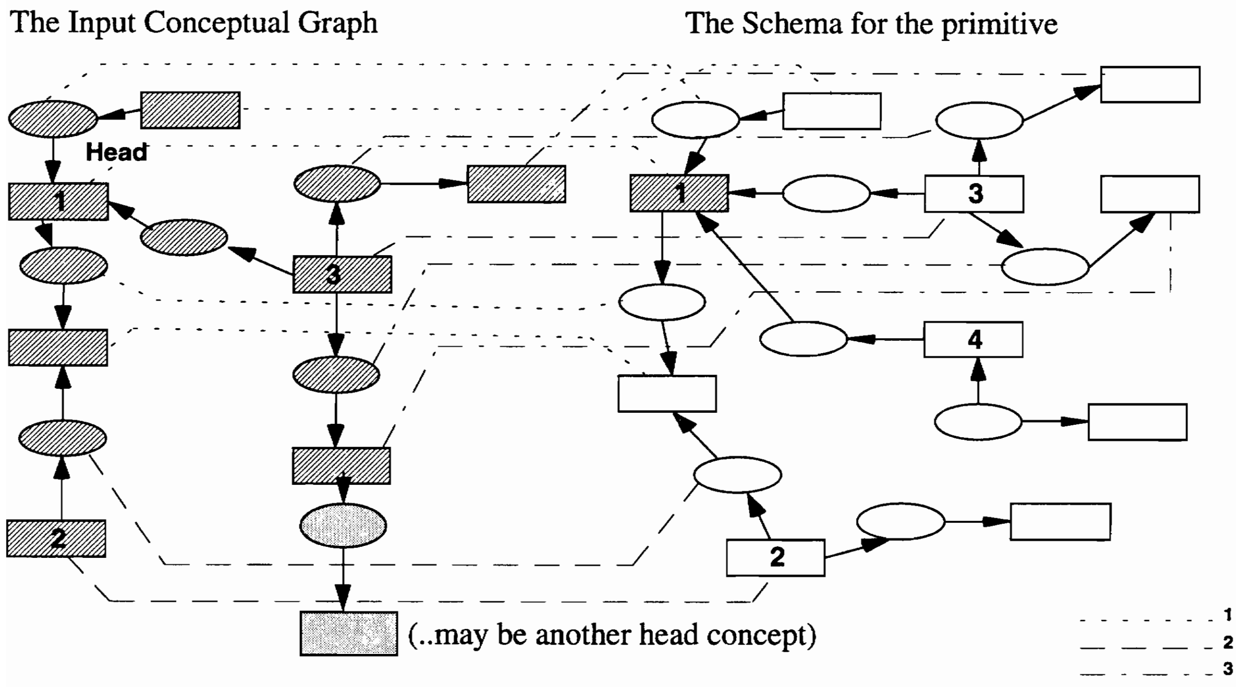

The Input Conceptual Graph
The Schema for the primitive
Head
1
3
2
1
3
4
2
(..may be another head concept)
1
2
3

Fig. 5.7 Projection arc between concepts and relations

An exception to Mugnier-Chein's projection rules are made while considering concepts and relations of the type shown in Fig. 5.8.

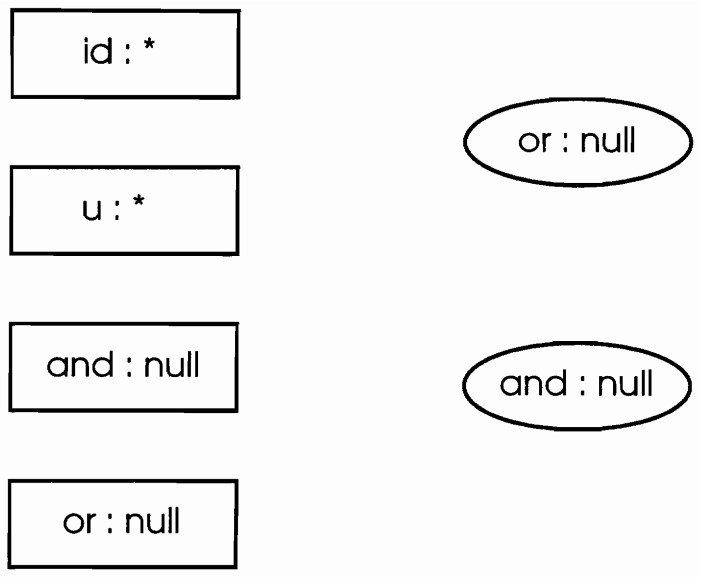


Fig. 5.8 Concept and relation types excluded from regular projection

The first of these concepts is of type *id* and represents a name type concept, i.e., signal names, variable names and generic names and usually input specification graphs have different names for theses concepts than the schema. In Fig. 5.9 SOUT and SREG are concepts of type id.

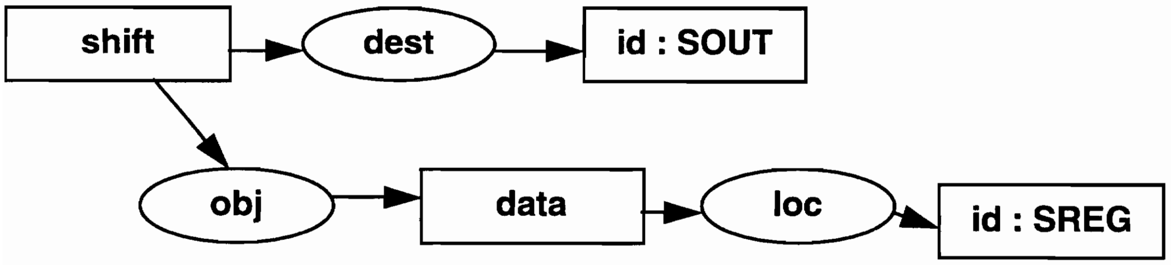


Fig. 5.9 Example of concepts of type id

Concepts of this type are not projected but are assumed to be correspondingly mapped. They are usually the last concept in a depth first search. These mappings are used while generating the final VHDL and the PMG, because they provide the new signal, variable and generic names.

The other concept and relation types are associated with multiple action unions. Consider an action in a schema which takes place only if two conditions are true as shown in Fig. 5.10, where *u* is a concept related by a *cond* relation from some action concept.

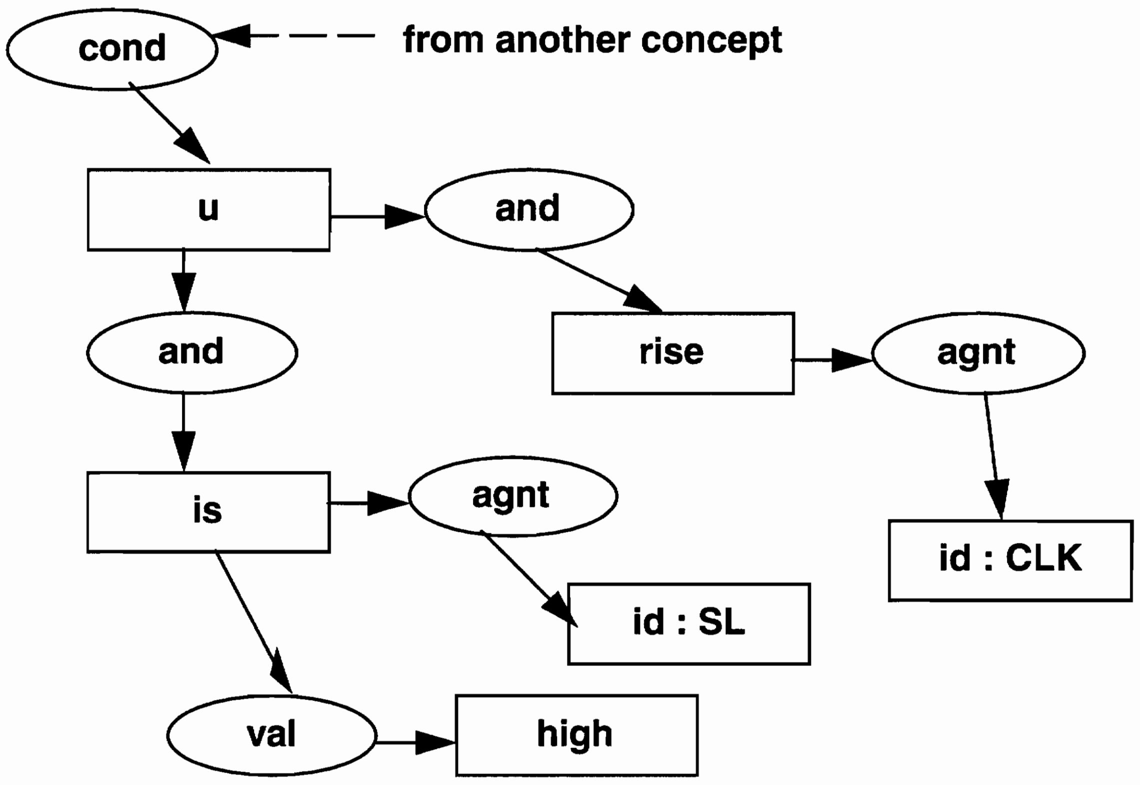


Fig. 5.10 Union of Multiple actions

The specification input graph may not specify all the conditions, e.g., in Fig. 5.10 the concept *is* and its outflowing arcs may not exist, hence the concept *u* will not exist as there is only one specified condition. However, a maximal projection is desired and so in this condition, the concept *u* and relations *and* and *or* are treated as being transparent, i.e., the projection operator continues through them as if they don't exist and continues to find one of the actions that may be specified, e.g., rise of the agent *CLK*. In such cases projection will succeed by mapping at least one of the two conditions rather than stopping the projection when the *u* concept is reached.

(iv) **From the graph G instantiate a subgraph S with the primitive concept at the head and connecting all the related concepts. Mark all concepts used in S as used.**

Shown in Fig. 5.11 is a session of mapping the Shift-Register schema on the input conceptual graph.

(4) shift-register	(1) shift-register
(6) OREG	(2) Q
(1) load	(3) load
(18) shift	(4) shift
(2) data	(5) data
(5) rise	(14) rise
(19) OP	(9) COUT
(17) rise	(15) rise
(3) DATA	(6) PARDAT
(7) LOAD	(8) LOAD
(13) clock	(11) clock
(26) OCLK	(12) CLK

Fig. 5.11 Mapped concepts

The concepts in the input conceptual graph used to generate a subgraph S which represents the projection of the schema in the input, i.e.,

$$S = \Pi (G)$$

are marked as used by setting an integer field in their data structures.

```
[  1 : device : shift-register ]
            ->( name : null )  -> [ 1:2 ]
[  2 : id : OREG ]
[  3 : write : load ]
            ->( gindx : 1 )  -> [  ]
            ->( obj : null )  -> [ 1:5 ]
            ->( dest : into )  -> [ 1:1 ]
            ->( cond : when )  -> [ 1:7 ]
[  4 : operate : shift ]
           ->( obj : null )  -> [ 1:2 ]
           ->( dest : in )  -> [ 1:9 ]
           ->( cond : when )  -> [ 1:10 [ 2:4 ] [ 3:9 ] ]
[  5 : data : data ]
            ->( name : null )  -> [ 1:6 ]
[  6 : id : DATA ]
[  7 : event : rise ]
            ->( agnt : null )  -> [ 1:8 ]
[  8 : id : LOAD ]
[  9 : id : OP ]
[ 10 : event : rise ]
            ->( agnt : null )  -> [ 1:11 [ 2:1 ] [ 3:10 ] ]
[ 11 : device : clock ]
            ->( name : null )  -> [ 1:12 [ 2:2 ] [ 3:11 ] ]
[ 12 : id : OCLK ]
```

Fig. 5.12 The subgraph S for a shift-register

The subgraph S is also stored in a data structure of type graphtype. It can be observed that all the concept id's have been renumbered. Each concept now also has a coreference field which provides the information about the other subgraphs S_i which may be referring to the same concept in the input. It is this data which used to interconnect the signals between the various processes in the last stage of generating the complete PMG. A small section of the subgraph is shown below,

```
[ 10 : event : rise ]
        ->( agnt : null )  -> [ 1:11 [ 2:1 ] [ 3:10 ] ]
[ 11 : device : clock ]
        ->( name : null )  -> [ 1:12 [ 2:2 ] [ 3:11 ] ]

[ 12 : id : OCLK ]
```

where, concept 1:11 also appears as 2:1 and 3:10 The numbers before the colon specify the graph numbers and those after the colon are the concept numbers.

(v) The Matches of referents from the previous step are used to update the signal names, variable names and generics in the primitive VHDL. The unmatched names are then renamed by prompting the user.

This step reads in the MODAS primitive file and scans it for all signal, variable, and generic names. Then from the mapping list from a previous step it decides which of the needed names were unspecified in the informal specification. The unspecified names are then reassigned by prompting the user for new names, e.g., see Fig. 5.13. If the user types in a / then the program assumes the default name as in the MODAS primitive.

```
This concept is undefined SERDAT , please enter a new referent :
/
 This concept is undefined SR , please enter a new referent :
/
```

Fig 5.13 Updated referent list (contd..)

```
This concept is undefined SL , please enter a new referent :
/
 This concept is undefined SREG_DEL , please enter a new referent :
/
 This concept is undefined CLKTRIG , please enter a new referent :
1
 This concept is undefined LOADTRIG , please enter a new referent :
1
This concept is undefined SRTRIG , please enter a new referent :
1
 This concept is undefined SLTRIG , please enter a new referent :
1
                              Referent   List

                          Schema              Specification

                 0      PARDAT       ----->  DATA
                 1      SERDAT               SERDAT
                 2      CLK          ----->  OCLK
                 3      LOAD                 LOAD
                 4      SR                   SR
                 5      SL                   SL
                 6      Q            ----->  OREG
                 7      SREG_DEL             SREG_DEL
                 8      CLKTRIG              1
                 9      LOADTRIG             1
                 10     SRTRIG               1
                 11     SLTRIG               1
                 12     COUT         ----->  OP
```

Fig. 5.13 Updated referent List

Note that in Fig. 5.13 only the names with an arrow after them were specified in the input, the rest were provided by the user when prompted.

(vi) **From the updated referent list a prototype of a MODAS primitive process is instantiated. Also a report is generated on the VHDL code generated.**

The MODAS primitive file is read now for the VHDL code. All instances of the signals, variables and generics are modified using the updated referent list. Comments are added at the beginning of each process to warn the user of all the new changes from the primitive and the differences between the model and the primitive.

```
SHIFTREG : process (CLK)
-- Specification lacked data for projecting concepts
--        SREG_DEL
--        SR,SL
if CLK = '1' then
 if LOAD = '1' then
  Q <= PARDAT after SREG_DEL;
  COUT <= PARDAT( 7 ) after SREG_DEL;
 elsif (SR = '1' and SL /= '1') then
  Q <= SERDAT & Q( 7 downto 1 ) after SREG_DEL;
  COUT <= Q( 0 ) after SREG_DEL;
 elsif (SR /= '1' and SL = '1') then
  Q <= Q( 6 downto 0 ) & SERDAT after SREG_DEL;
 COUT <= Q( 7 ) after SREG_DEL;
end if;
end if;
end process SHIFTREG;
```

Fig. 5.14 Modified VHDL code

(vii) Repeat steps (iii)-(vi) for all concepts in the table still marked unused.

This maps all the rest of the primitives taking care that primitive identifiers are not mapped repeatedly.

(viii) Concepts which are still marked unused are to define a new graph S' which is transformed into VHDL through the old approach.

The new approach encompasses the old technique and can revert to it at the command of the user if needed. Hence all the concepts that are not used in mapping to the primitives are combined into a graph called S' and are then processed by function calls to the old algorithm.

(ix) Generate the process and entity definition files enabling the user to view the Process Model Graph in MODAS.

The various process subgraphs generated in step (iv) are used to instantiate a process data structure which holds data about the signals, variables, generics and the process names. A pointer to process data structures is added to a Process Model Graph (PMG) data structure. In step (iv) a subgraph S was generated for each projection and coreference data which was written to a file. This data is now used to define the signals for the Process Model Graph. This is done by searching every subgraph S for a concepts which are signal names, variables or generics. If the coreference field of the concept data structure has entries then an entry is added to the Process Model Graph (PMG) data structure specifying the other process it shares the signal with. The PMG data structure is discussed in detail in Appendix B. The final function calls reads these data structures to generate binary files needed to view the Process Model Graph in MODAS.

Note: All main function calls are documented in the Appendix B : Programmers Reference.

Chapter 6

Conclusion

6.1 Results

The ASPIN system is a very complex and ambitious project. The various interfaces to the ASPIN system are currently operational with many constraints on the input. These problems will be solved with further research on the implementation of the system. The VHDL Linker too, will need further development to operate on conceptual graphs of various types.

Previously the CGVHDL Linker required sentences that mentioned a single executing behavior in one sentence. The new version adopted a new algorithm and can detect concepts referring to primitive processes from a graph. This allows the user to specify English sentences which specify the various components and their connections. This simplifies the users task as he does not have to specify detailed behavior of some standard processes. The inability to recognize a structure was a limitation of the previous version.

Currently the CGVHDL Linker has been quite successful in its task. The limitations of the system depend on the language specification. The more detailed the specification the more accurate is the VHDL. The main examples used to test the CGVHDL Linker are :

(i) The transmitter block of a UART
(ii) A Simple Latch
(iii) The i8212 chip
(iv) A finite state machine

The specification and the program output for each of the examples are documented in Appendix C. Fig. 6.1 shows a table that gives an idea of the complexity of the models and the results obtained. The success of the projection algorithm can be seen from the results tabulated in Fig. 6.2. The number of unmapped concepts varies greatly down the table because the number of sentences referring to a primitive are fewer and they do not provide enough information to cover the schema. The state machine example uses sentences that describe it's logic and a device that uses it's output. If there was a schema for a state machine then most of the concepts describing it would have been mapped to the schema and the number of unmapped concepts would be reduced.

Model	Number of sentences	Number of concepts in the input graph	Number of primitives used	Number of processes
UART Tx	5	28	3	5
A Simple Latch	3	18	2	3
i8212	10	45	3	8
A State Machine	7	37	1	5

Fig. 6.1 Summary of Results

Model	Primitives referred	Concepts in schema	Number of mappings	Size of the model	Left over concepts
UART Tx	Shift Register	20	12	28	5
	Oscillator	13	5		
	Counter	31	7		
A simple Latch	Register	30	7	18	7
	Buffer	8	6		
i8212	Mux	7	5	45	25
	Register	30	12		
	Buffer	8	5		
A state machine	Counter	31	5	37	32

Fig. 6.2 Table of Projection results

A summary of the tests and results observed are enumerated below,

(i) Different methods of using concepts to refer to a primitive have been tried. The UART example is a case where primitive referring concepts are of type device and their names specify the primitive referred. The other examples refer to primitives by using verbs describing their behavior, e.g., switch (for a multiplexer), apply (for a buffer) or load (for a register).

(ii) Also tested was the ability for maximal mapping of the behavior specification. This case is found in *if* condition statements. The schema may contain many conditions for an action but the input specifies just one. In such a case the program has been successful in mapping the corresponding schema concepts to the input.

(iii) Through all the examples the CGVHDL Linker was able to identify common signals among different processes and use them to generate signals which interconnect the processes.

(iv) The UART example shows a case where more than one concept referred to the same primitive, i.e., a counter primitive is referred to by the device concept and by the increment concept. In this case the Linker was able to determine that they were referring to a single instance of a counter and not two separate ones.

(v) Another case tested in the UART example was when two concepts referred to two entirely different primitives, e.g., a shift-register is referred to by the device concept and a register is referred to by the load concept, but performed operations on a same set of signals as show below,

```
[1:device:shift-register]
        ->(output:null)->[ 2 ]
[2:id:SREG]
[3:write:load]
        ->(obj:null)->[ 4 ]
        ->(src:null)->[ 5 ]
        ->(dest:null)->[ 2 ]
```

In this case the Linker inferred that the load concept did not refer to another register but the load behavior of the shift-register and thus instantiated only one shift-register process.

(vi) The CGVHDL Linker cannot map conditions for an *if* statement in excess of those present in the primitive's schema. In such cases these excess concepts are added to the list of unused concepts, but even the previous algorithm implemented by Alex Honcharik cannot interpret them completely as the activity concepts for the condition concepts are absent.

(vii) Another problem observed in all the tests is that most of the signal, variables and generics are of type '*unknown*'. This is due to the input specifications that usually do not provide the type of the involved signals, variables or generics. These types must be changed during the post-processing stage of execution of the Linker. The same problem applies to defining the size of a bit vector. The CGVHDL Linker by default sets the size to zero and this must be changed during post-processing.

From the above results it was concluded that the key to efficient operation of the Linker is dependent on two factors,

i. the detail covered by the specification input.
ii. the information provided about a primitive in it's schema.

The first factor is user dependent and can change from one user to another. The second factor is important because the better the schema, the better are the results of projection, i.e., larger number of concepts match between the input graph and a schema.

6.2 Future Enhancements

The CGVHDL Linker depends on the library of schemata which corresponds to the MODAS primitives. If the user adds more primitives to the MODAS primitive library then schemata for them must be added to the Linker's library of schemata. Currently all to be schemata used are written by hand by an operator knowledgeable about conceptual graphs and VHDL. To further automate the ASPIN system an interface with MODAS can be developed which could translate VHDL to a schemata. This can be a difficult process if it is fully automated, thus an interface to a user will have to be provided which will aid the computer to put the schema together.

The CGVHDL Linker can be further extended to cover a large number of more complex devices like UART's, memories, FPU's, and CPU's. These higher level components can be used to develop a VHDL model of more complex systems. This would again mean generating a schema for each which can be a laborious task.

The current version of CGVHDL uses an algorithm based on the Mugnier - Chein projection algorithm. Further work could involve a study of other graph matching methods to optimize projection.

Bibliography

[Honcarik 93] Alexander Honcharik, Generation of VHDL from Conceptual Grpahs of Informal Specification, Master's Thesis, Virginia Tech, 1993.

[Mugnier-Chien 92] M. L. Mugnier and M. Chein, Polynomial Algoritms for Projection and Matching, 7th Annual Workshop, Las Cruces, NM, USA, July 1992

[Mugnier-Chien 93] M. L. Mugnier and M. Chein, Characterization and algorithmic recognition of canonical graphs, First International Conference on Conceptual Structures,ICCS'93, Quebec City, Canada, August 1993

[Myaeng-Lopez 91] Proceedings 6th Annual Workshop on Conceptual Graphs, Binghampton, New York, July 1991

[Sowa 84] John F. Sowa, Conceptual Strucures: Information Processing in Mind and Machine, Addison-Wesley Publishing co., Reading, MA 1984.

[Armstrong 93] J. R. Armstrong, Structured Logic Design with VHDL, Prentice Hall, 1993

APPENDIX A
Results

A1 Transmitter of a UART

The sentences describing the UART are listed below.

(i) A rise on LOAD loads the DATA input value into shift-register OREG and sets NINTO high and OCLKEN high.

(ii) If OCLKEN is set, the clock OCLK is enabled and the counter OCTR is reset to "0000".

(iii) OREG is shifted onto OP with a rise on OCLK.

(iv) OCTR is incremented with a rise on OCLK.

(v) When OCTR is "1000" NINTO is reset.

(v) When OCTR is "1000" OCLKEN is reset.

In this case the words used to refer to the primitive are the actual device names bieng referred to.

The input graph for the transmitter is as follows,

```
[ 1 : write : load ]
    ->( gindx : 1 )
    ->( obj : null ) -> [ 2 ]
    ->( dest : into ) -> [ 4 ]
    ->( cond : when ) -> [ 5 ]
[ 2 : data : data ]
    ->( name : null ) -> [ 3 ]
[ 3 : id : DATA ]
[ 4 : device : shift-register ]
    ->( name : null ) -> [ 6 ]
[ 5 : event : rise ]
    ->( agnt : null ) -> [ 7 ]
[ 6 : id : OREG ]
[ 7 : id : LOAD ]
[ 8 : event : set ]
    ->( obj : null ) -> [ 9 ]
    ->(cond : null) -> [ 5 ]
[ 9 : id : NINTO ]
[ 10 : event : set ]
    ->( agnt : null ) -> [ 11 ]
[ 11 : id : OCLKEN ]
[ 12 : action : enable ]
    ->( obj : null ) -> [ 13 ]
    ->( cond : when ) -> [ 10 ]
[ 13 : device : clock ]
    ->( name : null ) -> [ 26 ]
[ 14 : event : reset]
    ->( obj : null ) -> [ 15 ]
    ->( attr : null ) -> [ 16 ]
    ->( cond : when ) -> [ 28 ]
[ 15 : device : counter ]
    ->( name : null) -> [ 27 ]
[ 16 : constant : 0 ]
[ 17 : event : rise ]
    ->( agnt : null ) -> [ 13 ]
[ 18 : operate : shift ]
    ->( obj : null ) -> [ 6 ]
    ->( dest : in ) -> [ 19 ]
    ->( cond : when ) -> [ 17 ]
[ 19 : id : OP ]
[ 20 : event : increment ]
     ->( obj : null ) -> [ 15 ]
     ->( cond : when ) -> [ 17 ]
[ 21 : state : is ]
     ->( agnt : null ) -> [ 15 ]
     ->( attr : null ) -> [ 22 ]
[ 22 : constant : "1000" ]
[ 23 : u : and ]
     ->(and : null ) -> [ 24 ]
     ->(and : null ) -> [ 25 ]
[ 24 : event : reset ]
     ->(obj : null ) -> [ 9 ]
     ->(cond : when ) -> [ 21 ]
[ 25 : event : reset ]
     ->(obj : null ) -> [ 11 ]
     ->(cond : when) -> [ 21 ]
[ 26 : id : OCLK ]
[ 27 : id : OCTR ]
[ 28 : event : rise ]
     ->( obj : null ) -> [ 11 ]
```

The subgraphs for each identified primitive.

```
The primitive graph is ....        _gr_no = 1
[  1 : device : shift-register ]
        ->( name : null )  -> [ 1:2 ]
[  2 : id : OREG ]
[  3 : write : load ]
        ->( gindx : 1 )  -> [  ]
        ->( obj : null )  -> [ 1:5 ]
        ->( dest : into )  -> [ 1:1 ]
        ->( cond : when )  -> [ 1:7 ]
[  4 : operate : shift ]
        ->( obj : null )  -> [ 1:2 ]
        ->( dest : in )  -> [ 1:9 ]
        ->( cond : when )  -> [ 1:10 [ 2:4 ] [ 3:9 ] ]
[  5 : data : data ]
        ->( name : null )  -> [ 1:6 ]
[  6 : id : DATA ]
[  7 : event : rise ]
        ->( agnt : null )  -> [ 1:8 ]
[  8 : id : LOAD ]
[  9 : id : OP ]
[ 10 : event : rise ]
        ->( agnt : null )  -> [ 1:11 [ 2:1 ] [ 3:10 ] ]
```

```
[ 11 : device : clock ]
        ->( name : null ) -> [ 1:12 [ 2:2 ] [ 3:11 ] ]
[ 12 : id : OCLK ]

The primitive graph is ....        _gr_no = 2
[  1 : device : clock ]
        ->( name : null ) -> [ 2:2 [ 1:12 ] [ 3:11 ] ]
[  2 : id : OCLK ]
[  3 : action : enable ]
        ->( obj : null ) -> [ 2:1 [ 1:11 ] [ 3:10 ] ]
        ->( cond : when ) -> [ 2:5 ]
[  4 : event : rise ]
        ->( agnt : null ) -> [ 2:1 [ 1:11 ] [ 3:10 ] ]
[  5 : event : set ]
        ->( agnt : null ) -> [ 2:6 [ 3:8 ] ]
[  6 : id : OCLKEN ]

The primitive graph is ....        _gr_no = 3
[  1 : device : counter ]
        ->( name : null ) -> [ 3:2 ]
[  2 : id : OCTR ]
[  3 : event : reset ]
        ->( obj : null ) -> [ 3:1 ]
        ->( attr : null ) -> [ 3:6 ]
        ->( cond : when ) -> [ 3:7 ]
[  4 : event : increment ]
        ->( obj : null ) -> [ 3:1 ]
        ->( cond : when ) -> [ 3:9 [ 1:10 ] [ 2:4 ] ]
[  5 : state : is ]
        ->( agnt : null ) -> [ 3:1 ]
        ->( attr : null ) -> [ 3:12 ]
[  6 : constant : 0 ]
[  7 : event : rise ]
        ->( obj : null ) -> [ 3:8 [ 2:6 ] ]
[  8 : id : OCLKEN ]
[  9 : event : rise ]
        ->( agnt : null ) -> [ 3:10 [ 1:11 ] [ 2:1 ] ]
[ 10 : device : clock ]
        ->( name : null ) -> [ 3:11 [ 1:12 ] [ 2:2 ] ]
[ 11 : id : OCLK ]
[ 12 : constant : "1000" ]

The primitive graph is ....        _gr_no = 4          /* This is a graph of leftover
[  1 : event : set ]                                    concepts and not a MODAS
        ->( obj : null ) -> [ 4:2 ]                     primitive */
        ->( cond : null ) -> [ 1:7 ]
[  2 : id : NINTO ]
[  3 : u : and ]
        ->( and : null ) -> [ 4:4 ]
        ->( and : null ) -> [ 4:5 ]
[  4 : event : reset ]
```

```
        ->( obj : null )  -> [ 4:2 ]
        ->( cond : when )  -> [ 3:5 ]
[  5 : event : reset ]
        ->( obj : null )  -> [ 2:6 ]
        ->( cond : when )  -> [ 3:5 ]
```

The CGVHDL output file.

```
[  1 : write : load ]                                          /* The input
        ->( gindx : 1 )  -> [  ]                               graph */
        ->( obj : null )  -> [ 0:2 ]
        ->( dest : into )  -> [ 0:4 ]
        ->( cond : when )  -> [ 0:5 ]
[  2 : data : data ]
        ->( name : null )  -> [ 0:3 ]
[  3 : id : DATA ]
[  4 : device : shift-register ]
        ->( name : null )  -> [ 0:6 ]
[  5 : event : rise ]
        ->( agnt : null )  -> [ 0:7 ]
[  6 : id : OREG ]
[  7 : id : LOAD ]
[  8 : event : set ]
        ->( obj : null )  -> [ 0:9 ]
        ->( cond : null )  -> [ 0:5 ]
[  9 : id : NINTO ]
[ 10 : event : set ]
        ->( agnt : null )  -> [ 0:11 ]
[ 11 : id : OCLKEN ]
[ 12 : action : enable ]
        ->( obj : null )  -> [ 0:13 ]
        ->( cond : when )  -> [ 0:10 ]
[ 13 : device : clock ]
        ->( name : null )  -> [ 0:26 ]
[ 14 : event : reset ]
        ->( obj : null )  -> [ 0:15 ]
        ->( attr : null )  -> [ 0:16 ]
        ->( cond : when )  -> [ 0:28 ]
[ 15 : device : counter ]
        ->( name : null )  -> [ 0:27 ]
[ 16 : constant : 0 ]
[ 17 : event : rise ]
        ->( agnt : null )  -> [ 0:13 ]
[ 18 : operate : shift ]
        ->( obj : null )  -> [ 0:6 ]
        ->( dest : in )  -> [ 0:19 ]
        ->( cond : when )  -> [ 0:17 ]
[ 19 : id : OP ]
[ 20 : event : increment ]
        ->( obj : null )  -> [ 0:15 ]
        ->( cond : when )  -> [ 0:17 ]
[ 21 : state : is ]
        ->( agnt : null )  -> [ 0:15 ]
        ->( attr : null )  -> [ 0:22 ]
[ 22 : constant : 8 ]
[ 23 : u : and ]
        ->( and : null )  -> [ 0:24 ]
        ->( and : null )  -> [ 0:25 ]
[ 24 : event : reset ]
        ->( obj : null )  -> [ 0:9 ]
        ->( cond : when )  -> [ 0:21 ]
[ 25 : event : reset ]
        ->( obj : null )  -> [ 0:11 ]
        ->( cond : when )  -> [ 0:21 ]
[ 26 : id : OCLK ]
[ 27 : id : OCTR ]
[ 28 : event : rise ]
        ->( obj : null )  -> [ 0:11 ]

Activity List:

 0 : set1  (con: 8)
     -> obj  -> obj 0
     -> cond  -> act 1
 1 : rise  (con: 5)
     -> agnt  -> obj 1
 2 : and  (con: 23)
     -> and  -> act 3
     -> and  -> act 5
 3 : reset2  (con: 24)
     -> obj  -> obj 0
     -> cond  -> act 4
 4 : is  (con: 21)
     -> agnt  -> obj 2
     -> attr  -> obj 3
 5 : reset2  (con: 25)
     -> obj  -> obj 4
     -> cond  -> act 4



Object List:

 0 : NINTO  ( signal :  ) (con: 9)
     type=unknown  size=0
 1 : LOAD  ( signal :  ) (con: 7)
```

```
     type=unknown  size=0
  2 : OCTR  ( signal : counter ) (con: 15)
     type=unknown  size=0
  3 : 8  ( value : 8 ) (con: 22)
  4 : OCLKEN  ( signal :  ) (con: 11)
     type=unknown  size=0

 cond -> act links
  1 -> 0
  4 -> 3
  4 -> 5


/* Summary of results */

 process SET_1  (NINTO:out, *LOAD:in)
 if ((LOAD='1') and LOAD'event) then
 NINTO <= '1';
 end if;

 process SET0_1  (NINTO:out, *OCTR:in)
 if (OCTR=8) then
 NINTO <= '0';
 end if;

 process SET0_2  (OCLKEN:out, *OCTR:in)
 if (OCTR=8) then
 OCLKEN <= '0';
 end if;

 process SHIFTREG  (OP:inout, OREG:inout, SL:inout, SR:inout, LOAD:inout,
  SERDAT:inout, OCLK:inout, DATA:inout)

 process OSC  (OCLK:inout, OCLKEN:inout)

 process COUNTER  (SYNC:inout, LOAD:inout, UP:inout, RESET:inout, EN:inout,
  CLK:inout, DATA:inout, CNT:inout)

 sig 0 : LOAD (BIT[0])
 sig 1 : NINTO (BIT[0])
 sig 2 : OCTR (unknown[0])
 sig 3 : OCLKEN (BIT[0])
 sig 4 : OP (unknown[0])
 sig 5 : OREG (unknown[0])
 sig 6 : SL (unknown[0])
 sig 7 : SR (unknown[0])
 sig 8 : SERDAT (unknown[0])
 sig 9 : OCLK (unknown[0])
 sig 10 : DATA (unknown[0])
 sig 11 : SYNC (unknown[0])
 sig 12 : UP (unknown[0])
sig 13 : RESET (unknown[0])
sig 14 : EN (unknown[0])
sig 15 : CLK (unknown[0])
sig 16 : CNT (unknown[0])



process SET_1  (NINTO:out, *LOAD:in, *OCTR:in)
if ((LOAD='1') and LOAD'event) then
NINTO <= '1';
end if;
if (OCTR=8) then
NINTO <= '0';
end if;

process SET0_2  (OCLKEN:out, *OCTR:in)
if (OCTR=8) then
OCLKEN <= '0';
end if;

process SHIFTREG  (OP:inout, OREG:inout, SL:inout, SR:inout, LOAD:inout,
 SERDAT:inout, OCLK:inout, DATA:inout)

process OSC  (OCLK:inout, OCLKEN:inout)

process COUNTER  (SYNC:inout, LOAD:inout, UP:inout, RESET:inout, EN:inout,
CLK:inout, DATA:inout, CNT:inout)
```

The execution window output

```
cgvhdl uart.g

/* Result of projection */
/* Input concepts      Schema Concepts    */

 ( 4 ) shift-register   ( 1 ) shift-register
 ( 6 ) OREG     ( 2 ) Q
 ( 1 ) load     ( 3 ) load
 ( 18 ) shift   ( 4 ) shift
 ( 2 ) data     ( 5 ) data
 ( 5 ) rise     ( 14 ) rise
 ( 19 ) OP      ( 9 ) COUT
 ( 17 ) rise    ( 15 ) rise
 ( 3 ) DATA     ( 6 ) PARDAT
 ( 7 ) LOAD     ( 8 ) LOAD
 ( 13 ) clock   ( 11 ) clock
 ( 26 ) OCLK    ( 12 ) CLK

 This concept is undefined SERDAT , please enter a new referent :     /* prompts for
/                                                                     new signal
This concept is undefined SR , please enter a new referent :         names */
/
 This concept is undefined SL , please enter a new referent :
/
 This concept is undefined SREG_DEL , please enter a new referent :
/
 This concept is undefined CLKTRIG , please enter a new referent :
/
 This concept is undefined LOADTRIG , please enter a new referent :
/
 This concept is undefined SRTRIG , please enter a new referent :
/
 This concept is undefined SLTRIG , please enter a new referent :
/

0 PARDAT DATA                                                         /*updated lignal
1 SERDAT SERDAT                                                       name list */
2 CLK OCLK
3 LOAD LOAD
4 SR SR
5 SL SL
6 Q OREG
7 SREG_DEL SREG_DEL
8 CLKTRIG CLKTRIG
9 LOADTRIG LOADTRIG
10 SRTRIG SRTRIG
11 SLTRIG SLTRIG
```

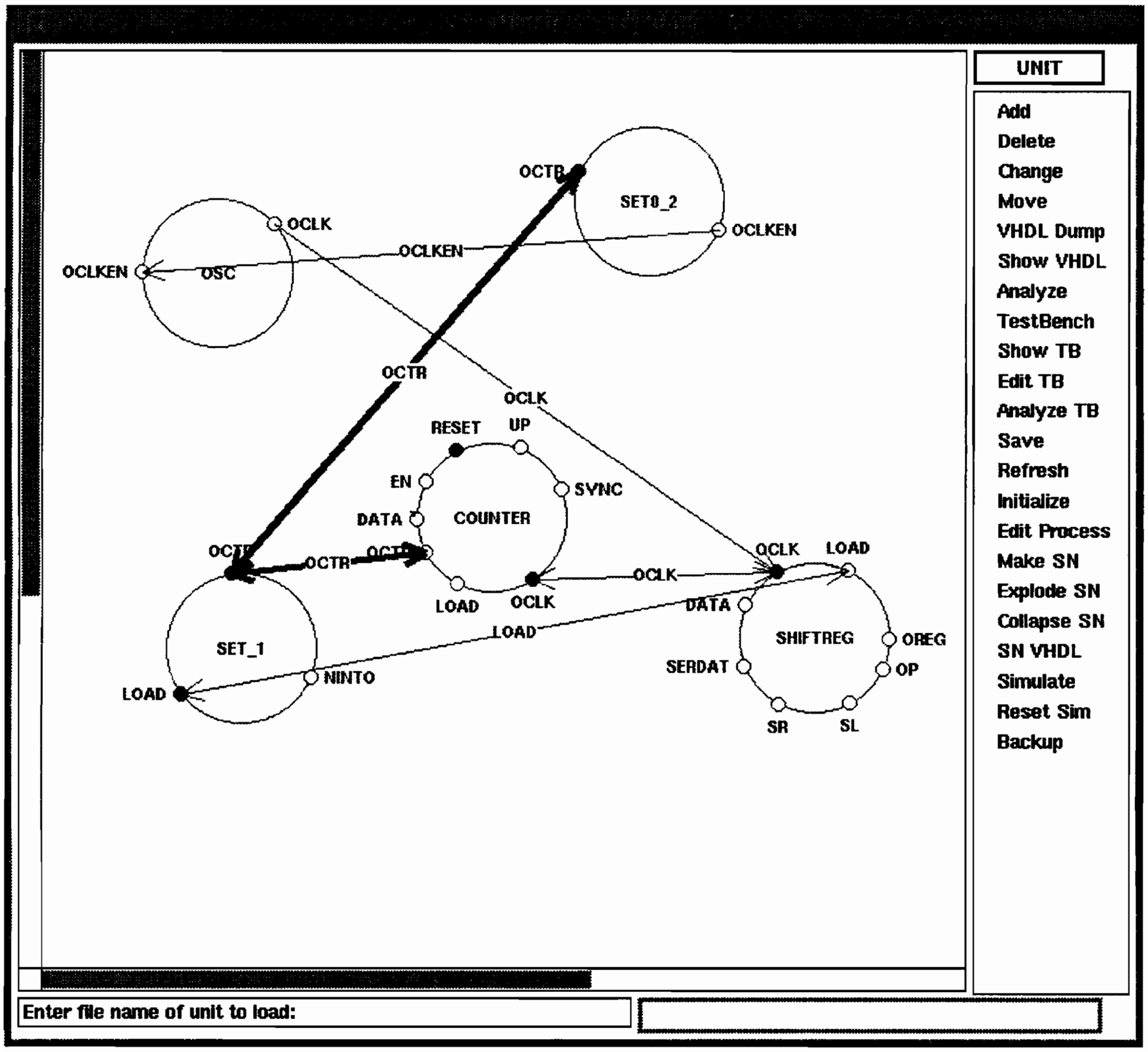
```
12 COUT OP

/* Modified VHDL code */

if OCLK = 'CLKTRIG' then
 if LOAD = 'LOADTRIG' then
  OREG <= DATA after SREG_DEL;
  OP <= DATA(#DATA.high) after SREG_DEL;
 elsif (SR = 'SRTRIG' and SL /= 'SLTRIG') then
  OREG <= SERDAT & OREG(#OREG.high downto #OREG.+1n) after SREG_DEL;
  OP <= SERDAT after SREG_DEL;
 elsif (SR /= 'SRTRIG' and SL = 'SLTRIG') then
  OREG <= OREG(#OREG.-1m downto #OREG.low) & SERDAT after SREG_DEL;
  OP <= OREG(#OREG.-1m) after SREG_DEL;
 end if
end if;

/* Result of projection */
/* Input concepts      Schema Concepts    */
 ( 13 ) clock    ( 1 ) clock
 ( 26 ) OCLK     ( 2 ) CLK
 ( 12 ) enable   ( 3 ) enable
 ( 10 ) set      ( 13 ) rise
 ( 11 ) OCLKEN   ( 6 ) CTRL

 This concept is undefined HITIME , please enter a new referent :      /*prompts for
/                                                                      new signal
 This concept is undefined LOTIME , please enter a new referent :      names */
/
 This concept is undefined CTRLTRIG , please enter a new referent :
/
0 CTRL OCLKEN                                                          /* updated signal
1 CLK OCLK                                                             name list */
2 HITIME HITIME
3 LOTIME LOTIME
4 CTRLTRIG CTRLTRIG

/* Modified VHDL code */

while OCLKEN = '1' loop
 OCLK <= '1';
 wait for HITIME;
 OCLK <= '0';
 wait for LOTIME;
end loop
wait until OCLKEN = '1';

 This concept is undefined DATA , please enter a new referent :
/
 This concept is undefined CLK , please enter a new referent :
/
```

```
 This concept is undefined EN , please enter a new referent :              /*promts for
/                                                                          new signal
 This concept is undefined RESET , please enter a new referent :           names */
/
 This concept is undefined UP , please enter a new referent :
/
 This concept is undefined LOAD , please enter a new referent :
/
 This concept is undefined SYNC , please enter a new referent :
/
 This concept is undefined CNT , please enter a new referent :
/
 This concept is undefined CNT_DEL , please enter a new referent :
/
 This concept is undefined CLKTRIG , please enter a new referent :
/
 This concept is undefined ENTRIG , please enter a new referent :
/
 This concept is undefined RESTTRIG , please enter a new referent :
/
 This concept is undefined UPTRIG , please enter a new referent :
/
 This concept is undefined LOADTRIG , please enter a new referent :
/
 This concept is undefined SYNCTRIG , please enter a new referent :
/

0 DATA DATA                                                                /* updated signal
1 CLK CLK                                                                  list */
2 EN EN
3 RESET RESET
4 UP UP
5 LOAD LOAD
6 SYNC SYNC
7 CNT CNT
8 CNT_DEL CNT_DEL
9 CLKTRIG CLKTRIG
10 ENTRIG ENTRIG
11 RESTTRIG RESTTRIG
12 UPTRIG UPTRIG
13 LOADTRIG LOADTRIG
14 SYNCTRIG SYNCTRIG

/*Modified VHDL code*/

if (RESET = '1' and RESET'EVENT and SYNC /= '1') or
  (RESET = '1' and SYNC = '1' and CLK = '1' and
   CLK'EVENT) then
 CNT <= "0000" after CNT_DEL;
elsif (LOAD = '1' and CLK = '1' and CLK'EVENT) then
 CNT <= DATA after CNT_DEL;
```

```
elsif (EN = '1' and CLK = '1' and CLK'EVENT) then
 if UP = '1' then
  CNT <= INC(CNT) after CNT_DEL;
 else
  CNT <= DEC(CNT) after CNT_DEL;
 end if
end if;

;    1    SHIFTREG     1                /* This is a list of all entries in the PMG
;    2    OP     2                         database.
;    4    OREG   2                          The last number in each row indicates
;    5    SL     2                          the entry type,
;    6    SR     2                                 1 - process name
;    7    LOAD   2                                 2 - signal name
;    8    SERDAT 2                                 3 - generic name          */
;    9    OCLK   2
;    10    DATA   2
;    3    SLTRIG 3
;    11    SRTRIG 3
;    12    LOADTRIG     3
;    13    CLKTRIG 3
;    14    SREG_DEL     3
;    14        0
;    14        0
;    1    OSC    1
;    2    OCLK   2
;    4    OCLKEN 2
;    3    LOTIME 3
;    5    HITIME 3
;    5        0
;    5        0
;    1    COUNTER 1
;    2    SYNC   2
;    4    LOAD   2
;    5    UP     2
;    6    RESET  2
;    7    EN     2
;    8    CLK    2
;    9    DATA   2
;    10    CNT    2
;    3    CNT_DEL 3
;    3        0
;    3        0

/* Summary of the PMG */

1 : process SET_1  (NINTO:out, *LOAD:in, *OCTR:in)
2 : process SET0_2  (OCLKEN:out, *OCTR:in)
3 : process SHIFTREG  (OP:inout, OREG:inout, SL:inout, SR:inout, LOAD:inout,
 SERDAT:inout, OCLK:inout, DATA:inout)
4 : process OSC  (OCLK:inout, OCLKEN:inout)
```

```
5 : process COUNTER  (SYNC:inout, LOAD:inout, UP:inout, RESET:inout, EN:inout,
 CLK:inout, DATA:inout, CNT:inout)
1 : sig 0 : LOAD (BIT[0])
2 : sig 1 : NINTO (BIT[0])
3 : sig 2 : OCTR (unknown[0])
4 : sig 3 : OCLKEN (BIT[0])
5 : sig 4 : OP (unknown[0])
6 : sig 5 : OREG (unknown[0])
7 : sig 6 : SL (unknown[0])
8 : sig 7 : SR (unknown[0])
9 : sig 8 : SERDAT (unknown[0])
10 : sig 9 : OCLK (unknown[0])
11 : sig 10 : DATA (unknown[0])
12 : sig 11 : SYNC (unknown[0])
13 : sig 12 : UP (unknown[0])
14 : sig 13 : RESET (unknown[0])
15 : sig 14 : EN (unknown[0])
16 : sig 15 : CLK (unknown[0])
17 : sig 16 : CNT (unknown[0])

(D)one, (A)bort writing, edit (S)ignal, (M)erge processes
change (P)rocess name, show (C)ode

d

warning!! file 'SET_1.mod' exists. overwrite? y

warning!! file 'SET0_2.mod' exists. overwrite? y

warning!! file 'uart.unt' exists. overwrite? y
```

The Process Model Graph



Above the OSC, SHIFTREG, and COUNTER processes are generated from primitives. Note the connecting signal are few due to inadequate speciciation in the sentences.

A2 A simple Latch

The sentences describing the latch are,

(i) If ENBLD is '1' then ***buffer*** REG to D0.

(ii) If DS1 is '1' and NDS2 os '0' then set ENBLD.

(iii) If CLK is '1' ***load*** REG with DATA.

Note in the above sentences, the words buffer and load refer to primitives of a BUFFER and a REGISTER in the MODAS primitive library. These words exist in a lookup file directing the LINKER to right primitive.

The input graph used was,

```
[ 1 : action :set]
     ->(obj:null)->[ 2 ]
     ->(cond : when)->[ 3 ]
[ 2 : id : ENBLD]
[ 3 : u : and]
     ->(and:null)->[ 4 ]
     ->(and:null)->[ 5 ]
[ 4 : state : is]
     ->(agnt:null)->[ 6 ]
     ->(attr:null)->[ 7 ]
[ 5 : state : is ]
     ->(agnt:null)->[ 8 ]
     ->(attr:null)->[ 9 ]
[ 6 : id : DS1]
[ 7 : constant : 1]
[ 8 : id : NDS2]
[ 9 : constant : 0]
[ 10 : action : load]
     ->(obj:null)->[ 11 ]
     ->(src:from)->[ 12 ]
      ->(cond when)->[ 13 ]
[ 11 : id : REG]
[ 12 : data : data]
```

```
        ->(name:null)->[ 14 ]
[ 13 : state : is]
        ->(agnt:null)->[ 15 ]
        ->(attr:null)->[ 7 ]
[ 14 : id : DATA]
[ 15 : id : CLK ]
[ 16 : action : buffer]
        ->( src : null)->[ 11 ]
        ->( dest : null)->[ 17 ]
        ->( cond : when)->[ 18 ]
[ 17 : id : DO]
[ 18 : state : is ]
        ->(agnt:null)->[ 2 ]
        ->(attr:null)->[ 7 ]
```

The subgraphs identified for this graph are,

```
The primitive graph is ....        _gr_no = 1
[  1 : action : load ]
          ->( obj : null )  -> [ 1:2 [ 2:2 ] ]
          ->( src : from )  -> [ 1:3 ]
          ->( cond : null )  -> [ 1:5 ]
[  2 : id : REG ]
[  3 : data : data ]
          ->( name : null )  -> [ 1:4 ]
[  4 : id : DATA ]
[  5 : state : is ]
          ->( agnt : null )  -> [ 1:6 ]
          ->( attr : null )  -> [ 1:7 [ 2:6 ] ]
[  6 : id : CLK ]
[  7 : constant : 1 ]

The primitive graph is ....        _gr_no = 2
[  1 : action : buffer ]
          ->( src : null )  -> [ 2:2 [ 1:2 ] ]
          ->( dest : null )  -> [ 2:3 ]
          ->( cond : when )  -> [ 2:4 ]
[  2 : id : REG ]
[  3 : id : DO ]
[  4 : state : is ]
          ->( agnt : null )  -> [ 2:5 ]
          ->( attr : null )  -> [ 2:6 [ 1:7 ] ]
[  5 : id : ENBLD ]
[  6 : constant : 1 ]
```

```
The primitive graph is ....       _gr_no = 3                /* This is a graph of
[  1 : action : set ]                                           unused concepts
        ->( obj : null )  -> [ 2:5 ]                            and not a MODAS
        ->( cond : when )  -> [ 3:2 ]                           primitive */
[  2 : u : and ]
        ->( and : null )  -> [ 3:3 ]
        ->( and : null )  -> [ 3:4 ]
[  3 : state : is ]
        ->( agnt : null )  -> [ 3:5 ]
        ->( attr : null )  -> [ 1:7 ]
[  4 : state : is ]
        ->( agnt : null )  -> [ 3:6 ]
        ->( attr : null )  -> [ 3:7 ]
[  5 : id : DS1 ]
[  6 : id : NDS2 ]
[  7 : constant : 0 ]
```

The CGVHDL output file

```
[  1 : action : set ]
        ->( obj : null )  -> [ 0:2 ]
        ->( cond : when )  -> [ 0:3 ]
[  2 : id : ENBLD ]
[  3 : u : and ]
        ->( and : null )  -> [ 0:4 ]
        ->( and : null )  -> [ 0:5 ]
[  4 : state : is ]
        ->( agnt : null )  -> [ 0:6 ]
        ->( attr : null )  -> [ 0:7 ]
[  5 : state : is ]
        ->( agnt : null )  -> [ 0:8 ]
        ->( attr : null )  -> [ 0:9 ]
[  6 : id : DS1 ]
[  7 : constant : 1 ]
[  8 : id : NDS2 ]
[  9 : constant : 0 ]
[ 10 : action : load ]
        ->( obj : null )  -> [ 0:11 ]
        ->( src : from )  -> [ 0:12 ]
        ->( cond : null )  -> [ 0:13 ]
[ 11 : id : REG ]
[ 12 : data : data ]
        ->( name : null )  -> [ 0:14 ]
[ 13 : state : is ]
        ->( agnt : null )  -> [ 0:15 ]
        ->( attr : null )  -> [ 0:7 ]
[ 14 : id : DATA ]
[ 15 : id : CLK ]
[ 16 : action : buffer ]
        ->( src : null )  -> [ 0:11 ]
        ->( dest : null )  -> [ 0:17 ]
        ->( cond : when )  -> [ 0:18 ]
[ 17 : id : DO ]
[ 18 : state : is ]
        ->( agnt : null )  -> [ 0:2 ]
        ->( attr : null )  -> [ 0:7 ]
```

```
Activity List:

 0 : set1  (con: 1)
      -> obj  -> obj 0
      -> cond  -> Mact 1
 1 : and  (con: 3)
      -> and  -> act 2
      -> and  -> act 3
 2 : is  (con: 4)
      -> agnt  -> obj 1
      -> attr  -> obj 2
 3 : is  (con: 5)
      -> agnt  -> obj 3
      -> attr  -> obj 4

Object List:

 0 : ENBLD  ( signal :  ) (con: 2)
      type=unknown  size=0
 1 : DS1  ( signal :  ) (con: 6)
      type=unknown  size=0
 2 : 1  ( value : 1 ) (con: 7)
 3 : NDS2  ( signal :  ) (con: 8)
      type=unknown  size=0
 4 : 0  ( value : 0 ) (con: 9)

cond -> act links
  1 ->  0


/*Summary of results*/

process SET_1  (ENBLD:out, *DS1:in, *NDS2:in)
if ((DS1=1) and (NDS2=0)) then
ENBLD <= '1';
end if;

process REGISTER  (REG:inout, CLK:inout, SYNC:inout, RESET:inout, CLK:inout,
 DATA:inout)

process BUFFER_V  (DO:inout, ENBLD:inout, REG:inout)

sig 0 : DS1 (unknown[0])
sig 1 : NDS2 (unknown[0])
sig 2 : ENBLD (BIT[0])
sig 3 : REG (unknown[0])
sig 4 : CLK (unknown[0])
sig 5 : SYNC (unknown[0])
sig 6 : RESET (unknown[0])
sig 7 : DATA (unknown[0])
sig 8 : DO (unknown[0])
```

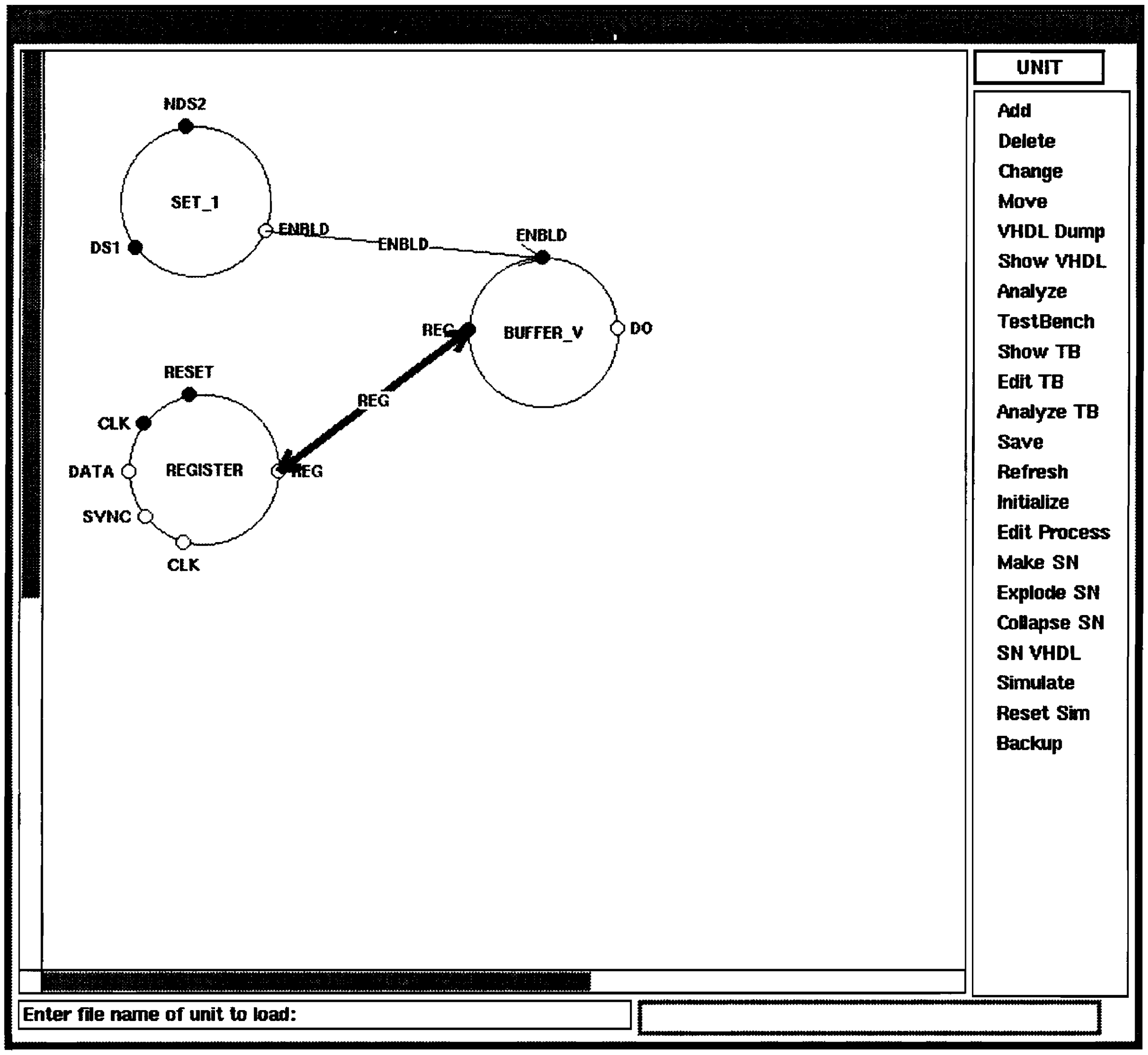
```
process SET_1  (ENBLD:out, *DS1:in, *NDS2:in)
if ((DS1=1) and (NDS2=0)) then
ENBLD <= '1';
end if;

process REGISTER  (REG:inout, CLK:inout, SYNC:inout, RESET:inout, CLK:inout,
 DATA:inout)

process BUFFER_V  (DO:inout, ENBLD:inout, REG:inout)
```

The execution window output

```
/* Result of projection */
/* Input concepts    Schema concepts */
     ( 10 ) load     ( 22 ) load
     ( 11 ) REG      ( 2 ) Q
     ( 12 ) data     ( 23 ) data
     ( 13 ) is       ( 26 ) is
     ( 14 ) DATA     ( 25 ) DATA
     ( 15 ) CLK      ( 27 ) LOAD
     ( 7 ) 1         ( 17 ) 1
     This concept is undefined RESET , please enter a new referent :      /* prompts to
    /                                                                      change signal
     This concept is undefined CLK , please enter a new referent :        names */
    /
     This concept is undefined SYNC , please enter a new referent :
    /
     This concept is undefined REG_DEL , please enter a new referent :
    /
     This concept is undefined CLKTRIG , please enter a new referent :
    1
     This concept is undefined RESTTRIG , please enter a new referent :
    1
     This concept is undefined LOADTRIG , please enter a new referent :
    1
     This concept is undefined SYNCTRIG , please enter a new referent :
    1
    0 DATA DATA                                                            /* updated
    1 LOAD CLK                                                             signal list */
    2 RESET RESET
    3 CLK CLK
    4 SYNC SYNC
    5 Q REG
    6 REG_DEL REG_DEL
```

```
7 CLKTRIG 1
8 RESTTRIG 1
9 LOADTRIG 1
10 SYNCTRIG 1

/* Modified VHDL code */

if (CLK = '1' and CLK'EVENT and RESET /= '1' and
   CLK = '1') then
 REG <= DATA after REG_DEL;
elsif (CLK = '1' and CLK'EVENT and SYNC = '1' and
      RESET = '1') or (SYNC /= '1' and RESET = '1'
      and RESET'EVENT) then
 REG <= "0000" after REG_DEL;
end if;

/* Results of projection */
/*Input concepts  Schema concepts */
 ( 16 ) buffer   ( 3 ) buffer
 ( 11 ) REG      ( 4 ) BUFIN
 ( 17 ) DO       ( 2 ) BUFOUT
 ( 18 ) is       ( 5 ) is
 ( 2 ) ENBLD     ( 6 ) EN
 ( 7 ) 1         ( 7 ) 1
 This concept is undefined IMP_DEL , please enter a new referent :     /* prompts for
/                                                                      changing signal
 This concept is undefined BUF_DEL , please enter a new referent :     names */
/

0 BUFIN REG                                                            /* updated
1 EN ENBLD                                                             signal list */
2 BUFOUT DO
3 IMP_DEL IMP_DEL
4 BUF_DEL BUF_DEL
if ENBLD = '1' then
 DO <= BVtoMVL4V(BUFIN) after BUF_DEL;
else
 DO <= "ZZZZ" after IMP_DEL;
end if;

/* PMG database list */

;    2    REGISTER     1
;    3    REG    2
;    5    CLK    2
;    6    SYNC   2
;    7    RESET  2
;    8    CLK    2
;    9    DATA   2
;    13   REG_DEL 3
;    13        0
;      13          0
;      2     BUFFER_V       1
;      3     DO     2
;      5     ENBLD  2
;      6     REG    2
;      4     IMP_DEL 3
;      7     BUF_DEL 3
;      7           0
;      7           0

/* Summary of Results */

1 : process SET_1  (ENBLD:out, *DS1:in, *NDS2:in)
2 : process REGISTER  (REG:inout, CLK:inout, SYNC:inout, RESET:inout,
 CLK:inout, DATA:inout)
3 : process BUFFER_V  (DO:inout, ENBLD:inout, REG:inout)
1 : sig 0 : DS1 (unknown[0])
2 : sig 1 : NDS2 (unknown[0])
3 : sig 2 : ENBLD (BIT[0])
4 : sig 3 : REG (unknown[0])
5 : sig 4 : CLK (unknown[0])
6 : sig 5 : SYNC (unknown[0])
7 : sig 6 : RESET (unknown[0])
8 : sig 7 : DATA (unknown[0])
9 : sig 8 : DO (unknown[0])

(D)one, (A)bort writing, edit (S)ignal, (M)erge processes
change (P)rocess name, show (C)ode

d


warning!! file 'regis.unt' exists. overwrite? y
```

The Process Model Graph



Enter file name of unit to load:

In this PMG, the processes REGISTER and BUFFER_V are generated by the projection algorithm. The process SET_1 is generated by using the old algorithm *[Honcharik 93]*.

A3 The i8212 port chip

The sentences used to describe the i8212 are,

(i) S0 is '1' when NDS1 is '0' and DS2 is '1'.

(ii) MD switches STB and S0 to S1.

(iii) REG is buffered to DO when EN is '1'.

(iv) EN is '1' when MD is '1' or S0 is '1'.

(v) When S1 is high and NCLR is low the data DI is latched into REG.

(vi) If NCLR is '1' REG is reset.

(vii) S2 is '0' when NCLR is low or S0 is '1'.

(viii) SRQ is reset when STB falls and S2 is '1'.

(ix) SRQ is set when S2 is '0'.

(x) NINT is low wehn S0 is high or SRQ is low.

Here the word *switch* refers to a multiplexer. These sentences generate a graph shown below, In this example the word *buffered* refers to a buffer. It could be changed to *apply* or *applied* without affecting the result of the Linker.

The complete graph used is,

```
[1: event:set]
    ->(agnt:null)->[2]
    ->(cond:null)->[3]
[2: id:S0]
[3: u:and]
    ->(and:null)->[4]
    ->(and:null)->[5]
[4: state :is]
    ->(agnt:null)->[6]
```

```
     ->(attr:null)->[7]
[5: state :is]
     ->(agnt:null)->[8]
     ->(attr:null)->[9]
[6: id:NDS1]
[7: constant:0]
[8: id:DS2]
[9: constant:1]
[10: action:switch]
     ->(agnt:null)->[11]
     ->(src:null)->[12]
     ->(dest:null)->[22]
[11 : id:MD]
[12 : u:and]
     ->(and:null)->[13]
     ->(and:null)->[2]
[13 : id:STB]
[14 : action:buffer]
     ->(src:null)->[15]
     ->(cond:null)->[16]
     ->(dest:null)->[39]
[15: id :REG]
[16: state:is]
     ->(agnt:null)->[17]
     ->(cond:null)->[9]
[17: id:EN]
[18: event:set]
     ->(obj:null)->[17]
     ->(cond:null)->[40]
[19: u:and]
     ->(and:null)->[20]
     ->(and:null)->[21]
[20: state :is]
     ->(agnt:null)->[22]
     ->(attr:null)->[9]
[21: state :is]
     ->(agnt:null)->[23]
     ->(attr:null)->[7]
[22: id:S1]
[23: id:NCLR]
[24: action:reset]
     ->(obj:null)->[15]
     ->(cond:when)->[25]
[25: state :is]
     ->(agnt:null)->[23]
     ->(attr:null)->[9]
[26: action:set]
     ->(obj:null)->[27]
     ->(cond:null)->[28]
[27 : id:SRQ]
[28 : state :is]
```

```
     ->(agnt:null)->[29]
     ->(attr:null)->[7]
[29 : id:S2]
[30: event:reset]
     ->(obj:null)->[27]
     ->(cond:when)->[31]
[31: u:and]
     ->(and:null)->[32]
     ->(and:null)->[33]
[32: event:fall]
     ->(agnt:null)->[13]
[33: state :is]
     ->(agnt:null)->[29]
     ->(attr:null)->[9]
[34: state :is]
     ->(agnt:null)->[35]
     ->(attr:null)->[7]
     ->(cond:null)->[36]
[35: id:NINT]
[36: u:or]
     ->(or:null)->[37]
     ->(or:null)->[38]
[37: state :is]
     ->(agnt:null)->[2]
     ->(attr:null)->[9]
[38: state :is]
     ->(agnt:null)->[27]
     ->(attr:null)->[7]
[39:id:D0]
[40: u:and]
     ->(and:null)->[41
     ->(and:null)->[42]
[41: state :is]
     ->(agnt:null)->[11]
     ->(attr:null)->[9]
[42: state :is]
     ->(agnt:null)->[2]
     ->(attr:null)->[9]
[43:action : load]
     ->(obj:null)->[ 15 ]
      ->(src:from)->[ 45 ]
      ->(cond when)->[ 19 ]
[44:id:DI]
[45:data:data]
     ->(name:null)->[44]
```

The primitve graphs formed are

```
The primitive graph is ....        _gr_no = 1

[  1 : action : switch ]
          ->( agnt : null )  -> [ 1:2 ]
          ->( src : null )  -> [ 1:3 ]
          ->( dest : null )  -> [ 1:6 [ 3:7 ] ]
[  2 : id : MD ]
[  3 : u : and ]
          ->( and : null )  -> [ 1:4 ]
          ->( and : null )  -> [ 1:5 ]
[  4 : id : STB ]
[  5 : id : S0 ]
[  6 : id : S1 ]

The primitive graph is ....        _gr_no = 2
[  1 : action : buffer ]
          ->( src : null )  -> [ 2:2 [ 3:2 ] ]
          ->( cond : null )  -> [ 2:3 ]
          ->( dest : null )  -> [ 2:6 ]
[  2 : id : REG ]
[  3 : state : is ]
          ->( agnt : null )  -> [ 2:4 ]
          ->( cond : null )  -> [ 2:5 [ 3:8 ] ]
[  4 : id : EN ]
[  5 : constant : 1 ]
[  6 : id : D0 ]

The primitive graph is ....        _gr_no = 3
[  1 : action : load ]
          ->( obj : null )  -> [ 3:2 [ 2:2 ] ]
          ->( src : from )  -> [ 3:3 ]
          ->( cond : null )  -> [ 3:5 ]
[  2 : id : REG ]
[  3 : data : data ]
          ->( name : null )  -> [ 3:4 ]
[  4 : id : DI ]
[  5 : u : and ]
          ->( and : null )  -> [ 3:6 ]
          ->( and : null )  -> [ 3:9 ]
[  6 : state : is ]
          ->( agnt : null )  -> [ 3:7 [ 1:6 ] ]
          ->( attr : null )  -> [ 3:8 [ 2:5 ] ]
[  7 : id : S1 ]
[  8 : constant : 1 ]
[  9 : state : is ]
```

```
        ->( agnt : null )  -> [ 3:10 ]
        ->( attr : null )  -> [ 3:11 ]
[ 10 : id : NCLR ]
[ 11 : constant : 0 ]

The primitive graph is ....        _gr_no = 4                          /* This is a graph of
[  1 : event : set ]                                                      unused cocepts */
        ->( agnt : null )  -> [ 1:5 ]
        ->( cond : null )  -> [ 4:2 ]
[  2 : u : and ]
        ->( and : null )  -> [ 4:3 ]
        ->( and : null )  -> [ 4:4 ]
[  3 : state : is ]
        ->( agnt : null )  -> [ 4:5 ]
        ->( attr : null )  -> [ 3:11 ]
[  4 : state : is ]
        ->( agnt : null )  -> [ 4:6 ]
        ->( attr : null )  -> [ 2:5 ]
[  5 : id : NDS1 ]
[  6 : id : DS2 ]
[  7 : event : set ]
        ->( obj : null )  -> [ 2:4 ]
        ->( cond : null )  -> [ 4:23 ]
[  8 : action : reset ]
        ->( obj : null )  -> [ 2:2 ]
        ->( cond : when )  -> [ 4:9 ]
[  9 : state : is ]
        ->( agnt : null )  -> [ 3:10 ]
        ->( attr : null )  -> [ 2:5 ]
[ 10 : action : set ]
        ->( obj : null )  -> [ 4:11 ]
        ->( cond : null )  -> [ 4:12 ]
[ 11 : id : SRQ ]
[ 12 : state : is ]
        ->( agnt : null )  -> [ 4:13 ]
        ->( attr : null )  -> [ 3:11 ]
[ 13 : id : S2 ]
[ 14 : event : reset ]
        ->( obj : null )  -> [ 4:11 ]
        ->( cond : when )  -> [ 4:15 ]
[ 15 : u : and ]
        ->( and : null )  -> [ 4:16 ]
        ->( and : null )  -> [ 4:17 ]
[ 16 : event : fall ]
        ->( agnt : null )  -> [ 1:4 ]
[ 17 : state : is ]
        ->( agnt : null )  -> [ 4:13 ]
        ->( attr : null )  -> [ 2:5 ]
[ 18 : state : is ]
        ->( agnt : null )  -> [ 4:19 ]
        ->( attr : null )  -> [ 3:11 ]
        ->( cond : null )  -> [ 4:20 ]
[ 19 : id : NINT ]
[ 20 : u : or ]
        ->( or : null )  -> [ 4:21 ]
        ->( or : null )  -> [ 4:22 ]
[ 21 : state : is ]
        ->( agnt : null )  -> [ 1:5 ]
        ->( attr : null )  -> [ 2:5 ]
[ 22 : state : is ]
        ->( agnt : null )  -> [ 4:11 ]
        ->( attr : null )  -> [ 3:11 ]
[ 23 : u : and ]
        ->( and : null )  -> [ 4:24 ]
        ->( and : null )  -> [ 4:25 ]
[ 24 : state : is ]
        ->( agnt : null )  -> [ 1:2 ]
        ->( attr : null )  -> [ 2:5 ]
[ 25 : state : is ]
        ->( agnt : null )  -> [ 1:5 ]
        ->( attr : null )  -> [ 2:5 ]
```

The CGVHDL output file

```
[  1 : event : set ]
        ->( agnt : null )  -> [ 0:2 ]
        ->( cond : null )  -> [ 0:3 ]
[  2 : id : S0 ]
[  3 : u : and ]
        ->( and : null )  -> [ 0:4 ]
        ->( and : null )  -> [ 0:5 ]
[  4 : state : is ]
        ->( agnt : null )  -> [ 0:6 ]
        ->( attr : null )  -> [ 0:7 ]
[  5 : state : is ]
        ->( agnt : null )  -> [ 0:8 ]
        ->( attr : null )  -> [ 0:9 ]
[  6 : id : NDS1 ]
[  7 : constant : 0 ]
[  8 : id : DS2 ]
[  9 : constant : 1 ]
[ 10 : action : switch ]
        ->( agnt : null )  -> [ 0:11 ]
        ->( src : null )  -> [ 0:12 ]
        ->( dest : null )  -> [ 0:22 ]
[ 11 : id : MD ]
[ 12 : u : and ]
        ->( and : null )  -> [ 0:13 ]
        ->( and : null )  -> [ 0:2 ]
[ 13 : id : STB ]
[ 14 : action : buffer ]
        ->( src : null )  -> [ 0:15 ]
        ->( cond : null )  -> [ 0:16 ]
        ->( dest : null )  -> [ 0:39 ]
[ 15 : id : REG ]
[ 16 : state : is ]
        ->( agnt : null )  -> [ 0:17 ]
        ->( cond : null )  -> [ 0:9 ]
[ 17 : id : EN ]
[ 18 : event : set ]
        ->( obj : null )  -> [ 0:17 ]
        ->( cond : null )  -> [ 0:40 ]
[ 19 : u : and ]
        ->( and : null )  -> [ 0:20 ]
        ->( and : null )  -> [ 0:21 ]
[ 20 : state : is ]
        ->( agnt : null )  -> [ 0:22 ]
        ->( attr : null )  -> [ 0:9 ]
[ 21 : state : is ]
        ->( agnt : null )  -> [ 0:23 ]
        ->( attr : null )  -> [ 0:7 ]
[ 22 : id : S1 ]
[ 23 : id : NCLR ]
[ 24 : action : reset ]
        ->( obj : null )  -> [ 0:15 ]
        ->( cond : when )  -> [ 0:25 ]
[ 25 : state : is ]
        ->( agnt : null )  -> [ 0:23 ]
        ->( attr : null )  -> [ 0:9 ]
[ 26 : action : set ]
        ->( obj : null )  -> [ 0:27 ]
        ->( cond : null )  -> [ 0:28 ]
[ 27 : id : SRQ ]
[ 28 : state : is ]
        ->( agnt : null )  -> [ 0:29 ]
        ->( attr : null )  -> [ 0:7 ]
[ 29 : id : S2 ]
[ 30 : event : reset ]
        ->( obj : null )  -> [ 0:27 ]
        ->( cond : when )  -> [ 0:31 ]
[ 31 : u : and ]
        ->( and : null )  -> [ 0:32 ]
        ->( and : null )  -> [ 0:33 ]
[ 32 : event : fall ]
        ->( agnt : null )  -> [ 0:13 ]
[ 33 : state : is ]
        ->( agnt : null )  -> [ 0:29 ]
        ->( attr : null )  -> [ 0:9 ]
[ 34 : state : is ]
        ->( agnt : null )  -> [ 0:35 ]
        ->( attr : null )  -> [ 0:7 ]
        ->( cond : null )  -> [ 0:36 ]
[ 35 : id : NINT ]
[ 36 : u : or ]
        ->( or : null )  -> [ 0:37 ]
        ->( or : null )  -> [ 0:38 ]
[ 37 : state : is ]
        ->( agnt : null )  -> [ 0:2 ]
        ->( attr : null )  -> [ 0:9 ]
[ 38 : state : is ]
        ->( agnt : null )  -> [ 0:27 ]
        ->( attr : null )  -> [ 0:7 ]
[ 39 : id : D0 ]
[ 40 : u : and ]
        ->( and : null )  -> [ 0:41 ]
        ->( and : null )  -> [ 0:42 ]
[ 41 : state : is ]
        ->( agnt : null )  -> [ 0:11 ]
        ->( attr : null )  -> [ 0:9 ]
[ 42 : state : is ]
        ->( agnt : null )  -> [ 0:2 ]
        ->( attr : null )  -> [ 0:9 ]
[ 43 : action : load ]
        ->( obj : null )  -> [ 0:15 ]
        ->( src : from )  -> [ 0:45 ]
        ->( cond : null )  -> [ 0:19 ]
[ 44 : id : DI ]
[ 45 : data : data ]
        ->( name : null )  -> [ 0:44 ]


Activity List:

 0 : set  (con: 1)
     -> agnt  -> obj 0
     -> cond  -> Mact 1
 1 : and  (con: 3)
     -> and  -> act 2
     -> and  -> act 3
 2 : is  (con: 4)
     -> agnt  -> obj 1
     -> attr  -> obj 2
 3 : is  (con: 5)
     -> agnt  -> obj 3
     -> attr  -> obj 4
 4 : set1  (con: 18)
     -> obj  -> obj 5
     -> cond  -> Mact 5
 5 : and  (con: 40)
      -> and  -> act 6
      -> and  -> act 7
 6 : is  (con: 41)
      -> agnt  -> obj 6
      -> attr  -> obj 4
 7 : is  (con: 42)
      -> agnt  -> obj 0
      -> attr  -> obj 4
 8 : reset2  (con: 24)
      -> obj  -> obj 7
      -> cond  -> act 9
 9 : is  (con: 25)
      -> agnt  -> obj 8
      -> attr  -> obj 4
10 : set1  (con: 26)
      -> obj  -> obj 9
      -> cond  -> act 11
11 : is  (con: 28)
      -> agnt  -> obj 10
      -> attr  -> obj 2
12 : reset2  (con: 30)
      -> obj  -> obj 9
      -> cond  -> Mact 13
13 : and  (con: 31)
      -> and  -> act 14
      -> and  -> act 15
14 : fall  (con: 32)
      -> agnt  -> obj 11
15 : is  (con: 33)
      -> agnt  -> obj 10
      -> attr  -> obj 4
16 : logic  (con: 34)
      -> agnt  -> obj 12
      -> attr  -> obj 2
      -> cond  -> Mact 17
17 : or  (con: 36)
      -> and  -> act 18
      -> and  -> act 19
18 : is  (con: 37)
      -> agnt  -> obj 0
      -> attr  -> obj 4
19 : is  (con: 38)
      -> agnt  -> obj 9
      -> attr  -> obj 2

Object List:

 0 : S0  ( signal :  ) (con: 2)
      type=unknown  size=0
 1 : NDS1  ( signal :  ) (con: 6)
```

```
     type=unknown  size=0
 2 : 0  ( value : 0 ) (con: 7)
 3 : DS2  ( signal :  ) (con: 8)
     type=unknown  size=0
 4 : 1  ( value : 1 ) (con: 9)
 5 : EN  ( signal :  ) (con: 17)
     type=unknown  size=0
 6 : MD  ( signal :  ) (con: 11)
     type=unknown  size=0
 7 : REG  ( signal :  ) (con: 15)
     type=unknown  size=0
 8 : NCLR  ( signal :  ) (con: 23)
     type=unknown  size=0
 9 : SRQ  ( signal :  ) (con: 27)
     type=unknown  size=0
10 : S2  ( signal :  ) (con: 29)
     type=unknown  size=0
11 : STB  ( signal :  ) (con: 13)
     type=unknown  size=0
12 : NINT  ( signal :  ) (con: 35)
     type=unknown  size=0

cond -> act links
  1 ->  0
  5 ->  4
  9 ->  8
 11 -> 10
 13 -> 12
 17 -> 16

/*Summary of results */


process set_1  (*NDS1:in, *DS2:in)
if ((NDS1=0) and (DS2=1)) then
end if;

process SET_1  (EN:out, *MD:in, *S0:in)
if ((MD=1) and (S0=1)) then
EN <= '1';
end if;

process SET0_1  (REG:out, *NCLR:in)
if (NCLR=1) then
REG <= '0';
end if;

process SET_2  (SRQ:out, *S2:in)
if (S2=0) then
SRQ <= '1';
end if;
```

```
process SET0_2  (SRQ:out, *STB:in, *S2:in)
if (((STB='0') and STB'event) and (S2=1)) then
SRQ <= '0';
end if;

process LOGIC_1  (NINT:out, *S0:in, *SRQ:in)
if ((S0=1) or (SRQ=0)) then
NINT <= 0;
else
 NINT <= not 0;
end if;

process MUX_2to1_1  (S1:inout, MD:inout, S0:inout, STB:inout)

process BUFFER_V  (D0:inout, EN:inout, REG:inout)

process REGISTER  (REG:inout, S1:inout, SYNC:inout, NCLR:inout, CLK:inout, DI:inout)

sig 0 : NDS1 (unknown[0])
sig 1 : DS2 (unknown[0])
sig 2 : MD (unknown[0])
sig 3 : S0 (unknown[0])
sig 4 : EN (BIT[0])
sig 5 : NCLR (unknown[0])
sig 6 : REG (BIT[0])
sig 7 : S2 (unknown[0])
sig 8 : SRQ (BIT[0])
sig 9 : STB (BIT[0])
sig 10 : NINT (unknown[0])
sig 11 : S1 (unknown[0])
sig 12 : D0 (unknown[0])
sig 13 : SYNC (unknown[0])
sig 14 : CLK (unknown[0])
sig 15 : DI (unknown[0])


/* Summary of results after merging processes */

process set_1  (*NDS1:in, *DS2:in)
if ((NDS1=0) and (DS2=1)) then
end if;

process SET_1  (EN:out, *MD:in, *S0:in)
if ((MD=1) and (S0=1)) then
EN <= '1';
end if;

process SET0_1  (REG:out, *NCLR:in)
if (NCLR=1) then
REG <= '0';
```

```
end if;

process SET_2  (SRQ:out, *S2:in, *STB:in)
if (S2=0) then
SRQ <= '1';
end if;
if (((STB='0') and STB'event) and (S2=1)) then
SRQ <= '0';
end if;

process LOGIC_1  (NINT:out, *S0:in, *SRQ:in)
if ((S0=1) or (SRQ=0)) then
NINT <= 0;
else
 NINT <= not 0;
end if;

process MUX_2to1_1  (S1:inout, MD:inout, S0:inout, STB:inout)

process BUFFER_V  (D0:inout, EN:inout, REG:inout)

process REGISTER  (REG:inout, S1:inout, SYNC:inout, NCLR:inout, CLK:inout, DI:inout)
```

The execution window output

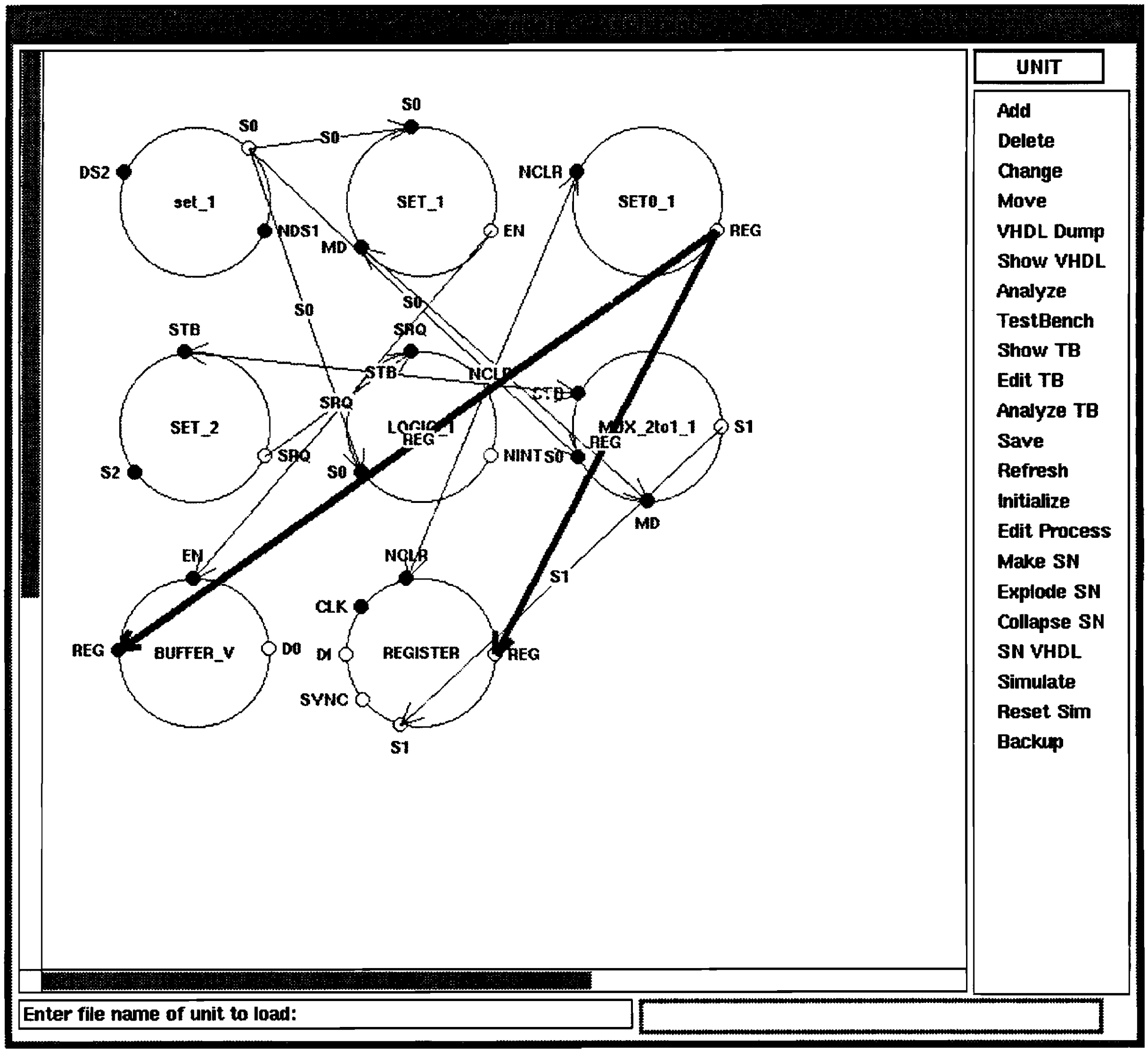
```
/* Result of projection */
/* Input concepts  Schema concepts */
 ( 10 ) switch     ( 1 ) switch
 ( 11 ) MD         ( 2 ) SEL
 ( 12 ) and         ( 3 ) and
 ( 22 ) S1         ( 4 ) Q
 ( 13 ) STB        ( 5 ) IN0
 ( 2 ) S0          ( 6 ) IN1
 This concept is undefined MUX_DEL , please enter a new referent :     /* prompt for
/                                                                      changing signal
                                                                       name */

0 IN0 STB                                                              /* updated signal
1 IN1 S0                                                               list */
2 SEL MD
3 Q S1
4 MUX_DEL MUX_DEL

/* modified VHDL code */

case MD is
  when '0' => S1 <= STB after MUX_DEL;
  when '1' => S1 <= S0 after MUX_DEL;
end case;

/* Results of projection */
/* Input concepts     Schema concepts */
 ( 14 ) buffer          ( 3 ) buffer
 ( 15 ) REG             ( 4 ) BUFIN
 ( 39 ) D0              ( 2 ) BUFOUT
 ( 16 ) is              ( 5 ) is
 ( 17 ) EN              ( 6 ) EN


 This concept is undefined IMP_DEL , please enter a new referent :     /*prompts for
/                                                                      changing signal
 This concept is undefined BUF_DEL , please enter a new referent :     name */
/


0 BUFIN REG                                                            /*updated signal
1 EN EN                                                                  list */
2 BUFOUT D0
3 IMP_DEL IMP_DEL
4 BUF_DEL BUF_DEL
```

```
/* modified VHDL code */

if EN = '1' then
 D0 <= BVtoMVL4V(BUFIN) after BUF_DEL;
else
 D0 <= "ZZZZ" after IMP_DEL;
end if;

/* Results for projection */
/* Input concepts        Schema concepts */
 ( 43 ) load             ( 22 ) load
 ( 15 ) REG              ( 2 ) Q
 ( 45 ) data             ( 23 ) data
 ( 19 ) and              ( 24 ) and
 ( 44 ) DI               ( 25 ) DATA
 ( 20 ) is      ( 26 ) is
 ( 21 ) is      ( 28 ) is
 ( 22 ) S1               ( 27 ) LOAD
 ( 9 ) 1        ( 17 ) 1
 ( 23 ) NCLR    ( 8 ) RESET
 ( 7 ) 0        ( 6 ) 0
 ( 25 ) is      ( 20 ) is


 This concept is undefined CLK , please enter a new referent :          /* prompts for
/                                                                       changing signal
 This concept is undefined SYNC , please enter a new referent :         names */
/
 This concept is undefined REG_DEL , please enter a new referent :
/
 This concept is undefined CLKTRIG , please enter a new referent :
1
 This concept is undefined RESTTRIG , please enter a new referent :
1
 This concept is undefined LOADTRIG , please enter a new referent :
1
 This concept is undefined SYNCTRIG , please enter a new referent :
1
0 DATA DI                                                               /* updated
1 LOAD S1                                                               signal list */
2 RESET NCLR
3 CLK CLK
4 SYNC SYNC
5 Q REG
6 REG_DEL REG_DEL
7 CLKTRIG 1
8 RESTTRIG 1
9 LOADTRIG 1
10 SYNCTRIG 1
```

```
/* modified VHDL code */

if (CLK = '1' and CLK'EVENT and NCLR /= '1' and
   S1 = '1') then
 REG <= DI after REG_DEL;
elsif (CLK = '1' and CLK'EVENT and SYNC = '1' and
      NCLR = '1') or (SYNC /= '1' and NCLR = '1'
      and NCLR'EVENT) then
 REG <= "0000" after REG_DEL;
end if;

/* PMG database */

;     8    MUX_2to1_1     1
;     4    S1     2
;     6    MD     2
;     5    S0     2
;     3    STB    2
;     7    MUX_DEL 3
;     7         0
;     7         0
;     2    BUFFER_V      1
;     3    D0     2
;     5    EN     2
;     6    REG    2
;     4    IMP_DEL 3
;     7    BUF_DEL 3
;     7         0
;     7         0
;     2    REGISTER      1
;     3    REG    2
;     5    S1     2
;     6    SYNC   2
;     7    NCLR   2
;     8    CLK    2
;     9    DI     2
;     13    REG_DEL 3
;     13        0
;     13        0

/* Summary of results */

1 : process set_1  (*NDS1:in, *DS2:in)
2 : process SET_1  (EN:out, *MD:in, *S0:in)
3 : process SET0_1  (REG:out, *NCLR:in)
4 : process SET_2  (SRQ:out, *S2:in, *STB:in)
5 : process LOGIC_1  (NINT:out, *S0:in, *SRQ:in)
6 : process MUX_2to1_1  (S1:inout, MD:inout, S0:inout, STB:inout)
7 : process BUFFER_V  (D0:inout, EN:inout, REG:inout)
8 : process REGISTER  (REG:inout, S1:inout, SYNC:inout, NCLR:inout, CLK:inout,
DI:inout)
```

```
1 : sig 0 : NDS1 (unknown[0])
2 : sig 1 : DS2 (unknown[0])
3 : sig 2 : MD (unknown[0])
4 : sig 3 : S0 (unknown[0])
5 : sig 4 : EN (BIT[0])
6 : sig 5 : NCLR (unknown[0])
7 : sig 6 : REG (BIT[0])
8 : sig 7 : S2 (unknown[0])
9 : sig 8 : SRQ (BIT[0])
10 : sig 9 : STB (BIT[0])
11 : sig 10 : NINT (unknown[0])
12 : sig 11 : S1 (unknown[0])
13 : sig 12 : D0 (unknown[0])
14 : sig 13 : SYNC (unknown[0])
15 : sig 14 : CLK (unknown[0])
16 : sig 15 : DI (unknown[0])

(D)one, (A)bort writing, edit (S)ignal, (M)erge processes
change (P)rocess name, show (C)ode

d

warning!! file 'SET0_1.mod' exists. overwrite? y

warning!! file 'LOGIC_1.mod' exists. overwrite? y

warning!! file 'i8212.unt' exists. overwrite? y
```

The Process Model Graph



Enter file name of unit to load:

In this graph the processes are MUX_2to1_1, BUFFER_V and REGISTER are generated by projection.

A4 A Simple state machine

The sentences used to describe the state machine are,

(i) When STATE is 1 and COIN is '1', STATE changes to 2.

(ii) When STATE is 2 and EN_MON is '0' , STATE changes to '1'.

(iii) If STATE is 2 and EN_MON is '1', STATE changes to 3.

(iv) If STATE is 3 change STATE to 1.

(v) If STATE is 2 then INCR is '1'.

(vi) If STATE is 3 then RESET is '1'.

(vi) If MON is "0101" then EN_MON is 1.

(vi) Increment MON if INCR is '1'.

(vii) Reset MON when RESET is 1.

The input conceptual graph formed was

```
[ 1 : write : change ]
      ->( dest : null ) -> [ 2 ]
      ->( attr : to ) -> [ 3 ]
      ->( cond : null ) -> [ 4 ]
[ 2 : id : STATE ]
[ 3 : constant : 2 ]
[ 4 : u : and ]
      ->( and : null ) -> [ 5 ]
      ->( and : null ) -> [ 7 ]
[ 5 : state : is ]
      ->( agnt : null ) -> [ 2 ]
      ->( attr : null ) -> [ 6 ]
[ 6 : constant : 1 ]
[ 7 : state : is ]
      ->( agnt : null ) -> [ 8 ]
      ->( attr : null ) -> [ 9 ]
[ 8 : id : COIN ]
[ 9 : constant : '1' ]
[ 10 : write : change ]
      ->( dest : null ) -> [ 2 ]
      ->( attr : to ) -> [ 6 ]
      ->( cond : null ) -> [ 11 ]
[ 11 : u : and ]
      ->( and : null ) -> [ 12 ]
      ->( and : null ) -> [ 13 ]
[ 12 : state : is ]
      ->( agnt : null ) -> [ 2 ]
      ->( attr : null ) -> [ 3 ]
[ 13 : state : is ]
      ->( agnt : null ) -> [ 14 ]
      ->( attr : null ) -> [ 21 ]
[ 14 : id : EN_MON ]
[ 15 : write : change ]
      ->( dest : null ) -> [ 2 ]
      ->( attr : to ) -> [ 20 ]
      ->( cond : null ) -> [ 16 ]
[ 16 : u : and ]
      ->( and : null ) -> [ 17 ]
      ->( and : null ) -> [ 18 ]
[ 17 : state : is ]
      ->( agnt : null ) -> [ 2 ]
      ->( attr : null ) -> [ 3 ]
[ 18 : state : is ]
      ->( agnt : null ) -> [ 14 ]
      ->( attr : null ) -> [ 9 ]
[ 19 : id : EN_MON ]
[ 20 : constant : 3]
[ 21 : constant : '0']
[ 22 : write : change ]
      ->( dest : null ) -> [ 2 ]
      ->( attr : to ) -> [ 6 ]
      ->( cond : null ) -> [ 23 ]
[ 23 : state : is ]
      ->( agnt : null ) -> [ 2 ]
      ->( attr : null ) -> [ 20 ]
[ 24 : state : is]
     ->( agnt : null ) -> [ 25 ]
     ->( attr : null ) -> [ 9 ]
      ->( cond : null ) -> [ 26 ]
[ 25 : id : INCR ]
[ 26 : state : is ]
     ->( agnt : null ) -> [ 2 ]
      ->( attr : null ) -> [ 3 ]
[ 27 : state : is ]
     ->( agnt : null ) -> [ 2 ]
      ->( attr : null ) -> [ 20 ]
[ 28 : id :null]
[ 29 : id :null]
[ 30 : action : increment ]
     ->( obj : null) -> [ 31 ]
     ->(cond : null) -> [ 32 ]
[ 31 : id : MON ]
[ 32 : state : is]
     ->( agnt : null) -> [ 25 ]
     ->( attr : null) -> [ 9 ]
[ 33 : action : reset ]
     ->( obj : null) -> [ 31 ]
     ->(cond : null) -> [ 34 ]
[ 34 : state : is]
     ->( agnt : null) -> [ 14 ]
     ->( attr : null) -> [ 9 ]
[ 35 : state : is]
     ->( agnt : null) -> [ 14 ]
     ->( attr : null) -> [ 9 ]
     ->( cond : when) -> [ 36 ]
[ 36 : state : is]
     ->( agnt : null) -> [ 31 ]
     ->( attr : null) -> [ 37 ]
[ 37 : constant : "0010"]
```

The primitive graphs formed are,

```
The primitive graph is ....        _gr_no = 1
[  1 : action : increment ]
        ->( obj : null )  -> [ 1:2 ]
        ->( cond : null )  -> [ 1:3 ]
[  2 : id : MON ]
[  3 : state : is ]
        ->( agnt : null )  -> [ 1:4 ]
        ->( attr : null )  -> [ 1:5 ]
[  4 : id : INCR ]
[  5 : constant : '1' ]

The primitive graph is ....        _gr_no = 2                       /* Graph of unused
[  1 : write : change ]                                                concepts */
        ->( dest : null )  -> [ 2:2 ]
        ->( attr : to )  -> [ 2:3 ]
        ->( cond : null )  -> [ 2:4 ]
[  2 : id : STATE ]
[  3 : constant : 2 ]
[  4 : u : and ]
        ->( and : null )  -> [ 2:5 ]
        ->( and : null )  -> [ 2:7 ]
[  5 : state : is ]
        ->( agnt : null )  -> [ 2:2 ]
        ->( attr : null )  -> [ 2:6 ]
[  6 : constant : 1 ]
[  7 : state : is ]
        ->( agnt : null )  -> [ 2:8 ]
        ->( attr : null )  -> [ 1:5 ]
[  8 : id : COIN ]
[  9 : write : change ]
        ->( dest : null )  -> [ 2:2 ]
        ->( attr : to )  -> [ 2:6 ]
        ->( cond : null )  -> [ 2:10 ]
[ 10 : u : and ]
        ->( and : null )  -> [ 2:11 ]
        ->( and : null )  -> [ 2:12 ]
[ 11 : state : is ]
        ->( agnt : null )  -> [ 2:2 ]
        ->( attr : null )  -> [ 2:3 ]
[ 12 : state : is ]
        ->( agnt : null )  -> [ 2:13 ]
        ->( attr : null )  -> [ 2:20 ]
[ 13 : id : EN_MON ]
[ 14 : write : change ]
        ->( dest : null )  -> [ 2:2 ]
        ->( attr : to )  -> [ 2:19 ]
        ->( cond : null )  -> [ 2:15 ]
[ 15 : u : and ]
        ->( and : null )  -> [ 2:16 ]
        ->( and : null )  -> [ 2:17 ]
[ 16 : state : is ]
        ->( agnt : null )  -> [ 2:2 ]
        ->( attr : null )  -> [ 2:3 ]
[ 17 : state : is ]
        ->( agnt : null )  -> [ 2:13 ]
        ->( attr : null )  -> [ 1:5 ]
[ 18 : id : EN_MON ]
[ 19 : constant : 3 ]
[ 20 : constant : '0' ]
[ 21 : write : change ]
        ->( dest : null )  -> [ 2:2 ]
        ->( attr : to )  -> [ 2:6 ]
        ->( cond : null )  -> [ 2:22 ]
[ 22 : state : is ]
        ->( agnt : null )  -> [ 2:2 ]
        ->( attr : null )  -> [ 2:19 ]
[ 23 : state : is ]
        ->( agnt : null )  -> [ 1:4 ]
        ->( attr : null )  -> [ 1:5 ]
        ->( cond : null )  -> [ 2:24 ]
[ 24 : state : is ]
        ->( agnt : null )  -> [ 2:2 ]
        ->( attr : null )  -> [ 2:3 ]
[ 25 : state : is ]
```

```
          ->( agnt : null )  -> [ 2:2 ]
          ->( attr : null )  -> [ 2:19 ]
[ 26 : id : null ]
[ 27 : id : null ]
[ 28 : action : reset ]
          ->( obj : null )  -> [ 1:2 ]
          ->( cond : null )  -> [ 2:29 ]
[ 29 : state : is ]
          ->( agnt : null )  -> [ 2:13 ]
          ->( attr : null )  -> [ 1:5 ]
[ 30 : state : is ]
          ->( agnt : null )  -> [ 2:13 ]
          ->( attr : null )  -> [ 1:5 ]
          ->( cond : when )  -> [ 2:31 ]
[ 31 : state : is ]
          ->( agnt : null )  -> [ 1:2 ]
          ->( attr : null )  -> [ 2:32 ]
[ 32 : constant : "0010" ]
```

The CGVHDL output file

```
[  1 : write : change ]
          ->( dest : null )  -> [ 0:2 ]
          ->( attr : to )  -> [ 0:3 ]
          ->( cond : null )  -> [ 0:4 ]
[  2 : id : STATE ]
[  3 : constant : 2 ]
[  4 : u : and ]
          ->( and : null )  -> [ 0:5 ]
          ->( and : null )  -> [ 0:7 ]
[  5 : state : is ]
          ->( agnt : null )  -> [ 0:2 ]
          ->( attr : null )  -> [ 0:6 ]
[  6 : constant : 1 ]
[  7 : state : is ]
          ->( agnt : null )  -> [ 0:8 ]
          ->( attr : null )  -> [ 0:9 ]
[  8 : id : COIN ]
[  9 : constant : '1' ]
[ 10 : write : change ]
          ->( dest : null )  -> [ 0:2 ]
          ->( attr : to )  -> [ 0:6 ]
          ->( cond : null )  -> [ 0:11 ]
[ 11 : u : and ]
          ->( and : null )  -> [ 0:12 ]
          ->( and : null )  -> [ 0:13 ]
```

```
[ 12 : state : is ]
        ->( agnt : null )  -> [ 0:2 ]
        ->( attr : null )  -> [ 0:3 ]
[ 13 : state : is ]
        ->( agnt : null )  -> [ 0:14 ]
        ->( attr : null )  -> [ 0:21 ]
[ 14 : id : EN_MON ]
[ 15 : write : change ]
        ->( dest : null )  -> [ 0:2 ]
        ->( attr : to )  -> [ 0:20 ]
        ->( cond : null )  -> [ 0:16 ]
[ 16 : u : and ]
        ->( and : null )  -> [ 0:17 ]
        ->( and : null )  -> [ 0:18 ]
[ 17 : state : is ]
        ->( agnt : null )  -> [ 0:2 ]
        ->( attr : null )  -> [ 0:3 ]
[ 18 : state : is ]
        ->( agnt : null )  -> [ 0:14 ]
        ->( attr : null )  -> [ 0:9 ]
[ 19 : id : EN_MON ]
[ 20 : constant : 3 ]
[ 21 : constant : '0' ]
[ 22 : write : change ]
        ->( dest : null )  -> [ 0:2 ]
        ->( attr : to )  -> [ 0:6 ]
        ->( cond : null )  -> [ 0:23 ]
[ 23 : state : is ]
        ->( agnt : null )  -> [ 0:2 ]
        ->( attr : null )  -> [ 0:20 ]
[ 24 : state : is ]
        ->( agnt : null )  -> [ 0:25 ]
        ->( attr : null )  -> [ 0:9 ]
        ->( cond : null )  -> [ 0:26 ]
[ 25 : id : INCR ]
[ 26 : state : is ]
        ->( agnt : null )  -> [ 0:2 ]
        ->( attr : null )  -> [ 0:3 ]
[ 27 : state : is ]
        ->( agnt : null )  -> [ 0:2 ]
        ->( attr : null )  -> [ 0:20 ]
[ 28 : id : null ]
[ 29 : id : null ]
[ 30 : action : increment ]
        ->( obj : null )  -> [ 0:31 ]
        ->( cond : null )  -> [ 0:32 ]
[ 31 : id : MON ]
[ 32 : state : is ]
        ->( agnt : null )  -> [ 0:25 ]
        ->( attr : null )  -> [ 0:9 ]
[ 33 : action : reset ]
```

```
        ->( obj : null )  -> [ 0:31 ]
        ->( cond : null )  -> [ 0:34 ]
[ 34 : state : is ]
        ->( agnt : null )  -> [ 0:14 ]
        ->( attr : null )  -> [ 0:9 ]
[ 35 : state : is ]
        ->( agnt : null )  -> [ 0:14 ]
        ->( attr : null )  -> [ 0:9 ]
        ->( cond : when )  -> [ 0:36 ]
[ 36 : state : is ]
        ->( agnt : null )  -> [ 0:31 ]
        ->( attr : null )  -> [ 0:37 ]
[ 37 : constant : "0010" ]
```

Activity List:

```
0 : change1  (con: 1)
     -> dest  -> obj 0
     -> attr  -> obj 1
     -> cond  -> Mact 1
1 : and  (con: 4)
     -> and  -> act 2
     -> and  -> act 3
2 : is  (con: 5)
     -> agnt  -> obj 0
     -> attr  -> obj 2
3 : is  (con: 7)
     -> agnt  -> obj 3
     -> attr  -> obj 4
4 : change1  (con: 10)
     -> dest  -> obj 0
     -> attr  -> obj 2
     -> cond  -> Mact 5
5 : and  (con: 11)
     -> and  -> act 6
     -> and  -> act 7
6 : is  (con: 12)
     -> agnt  -> obj 0
     -> attr  -> obj 1
7 : is  (con: 13)
     -> agnt  -> obj 5
     -> attr  -> obj 6
8 : change1  (con: 15)
     -> dest  -> obj 0
     -> attr  -> obj 7
     -> cond  -> Mact 9
9 : and  (con: 16)
     -> and  -> act 10
     -> and  -> act 11
10 : is  (con: 17)
     -> agnt  -> obj 0
     -> attr  -> obj 1
11 : is  (con: 18)
     -> agnt  -> obj 5
     -> attr  -> obj 4
12 : change1  (con: 22)
     -> dest  -> obj 0
     -> attr  -> obj 2
     -> cond  -> act 13
13 : is  (con: 23)
     -> agnt  -> obj 0
     -> attr  -> obj 7
14 : logic  (con: 24)
     -> agnt  -> obj 8
     -> attr  -> obj 4
     -> cond  -> act 15
15 : is  (con: 26)
     -> agnt  -> obj 0
     -> attr  -> obj 1
16 : is  (con: 27)
     -> agnt  -> obj 0
     -> attr  -> obj 7
17 : reset2  (con: 33)
     -> obj  -> obj 9
     -> cond  -> act 18
18 : is  (con: 34)
     -> agnt  -> obj 5
     -> attr  -> obj 4
19 : logic  (con: 35)
     -> agnt  -> obj 5
     -> attr  -> obj 4
     -> cond  -> act 20
20 : is  (con: 36)
     -> agnt  -> obj 9
     -> attr  -> obj 10

Object List:

0 : STATE  ( signal :  ) (con: 2)
    type=unknown  size=0
1 : 2  ( value : 2 ) (con: 3)
2 : 1  ( value : 1 ) (con: 6)
3 : COIN  ( signal :  ) (con: 8)
    type=unknown  size=0
4 : '1'  ( value : '1' ) (con: 9)
5 : EN_MON  ( signal :  ) (con: 14)
    type=unknown  size=0
6 : '0'  ( value : '0' ) (con: 21)
7 : 3  ( value : 3 ) (con: 20)
```

```
 8 : INCR  ( signal :  ) (con: 25)
      type=unknown  size=0
 9 : MON  ( signal :  ) (con: 31)
      type=unknown  size=0
10 : "0010"  ( value : "0010" ) (con: 37)

cond -> act links
  1 ->  0
  5 ->  4
  9 ->  8
 13 -> 12
 15 -> 14
 18 -> 17
 20 -> 19


/* Result Summary */

process CHANGE_1  (*STATE:inout, *COIN:in)
if ((STATE=1) and (COIN='1')) then
STATE <= 2;
end if;

process CHANGE_2  (*STATE:inout, *EN_MON:in)
if ((STATE=2) and (EN_MON='0')) then
STATE <= 1;
end if;

process CHANGE_3  (*STATE:inout, *EN_MON:in)
if ((STATE=2) and (EN_MON='1')) then
STATE <= 3;
end if;

process CHANGE_4  (*STATE:inout)
if (STATE=3) then
STATE <= 1;
end if;

process LOGIC_1  (INCR:out, *STATE:in)
if (STATE=2) then
INCR <= '1';
else
 INCR <= not '1';
end if;

process SET0_1  (MON:out, *EN_MON:in)
if (EN_MON='1') then
MON <= '0';
end if;

process LOGIC_2  (EN_MON:out, *MON:in)
```

```
if (MON="0010") then
EN_MON <= '1';
else
  EN_MON <= not '1';
end if;

process COUNTER  (SYNC:inout, LOAD:inout, UP:inout, RESET:inout, EN:inout,
CLK:inout, DATA:inout, MON:inout)

sig 0 : STATE (unknown[0])
sig 1 : COIN (unknown[0])
sig 2 : EN_MON (unknown[0])
sig 3 : INCR (unknown[0])
sig 4 : MON (BIT[0])
sig 5 : SYNC (unknown[0])
sig 6 : LOAD (unknown[0])
sig 7 : UP (unknown[0])
sig 8 : RESET (unknown[0])
sig 9 : EN (unknown[0])
sig 10 : CLK (unknown[0])
sig 11 : DATA (unknown[0])


/* Summary of Results after merging processes */

process CHANGE_1  (*STATE:inout, *COIN:in, *EN_MON:in)
if ((STATE=1) and (COIN='1')) then
STATE <= 2;
end if;
if (STATE=3) then
STATE <= 1;
end if;
if ((STATE=2) and (EN_MON='1')) then
STATE <= 3;
end if;
if ((STATE=2) and (EN_MON='0')) then
STATE <= 1;
end if;

process LOGIC_1  (INCR:out, *STATE:in)
if (STATE=2) then
INCR <= '1';
else
  INCR <= not '1';
end if;

process SET0_1  (MON:out, *EN_MON:in)
if (EN_MON='1') then
MON <= '0';
end if;
```

```
process LOGIC_2  (EN_MON:out, *MON:in)
if (MON="0010") then
EN_MON <= '1';
else
 EN_MON <= not '1';
end if;

process COUNTER  (SYNC:inout, LOAD:inout, UP:inout, RESET:inout, EN:inout,
 CLK:inout, DATA:inout, MON:inout)
```

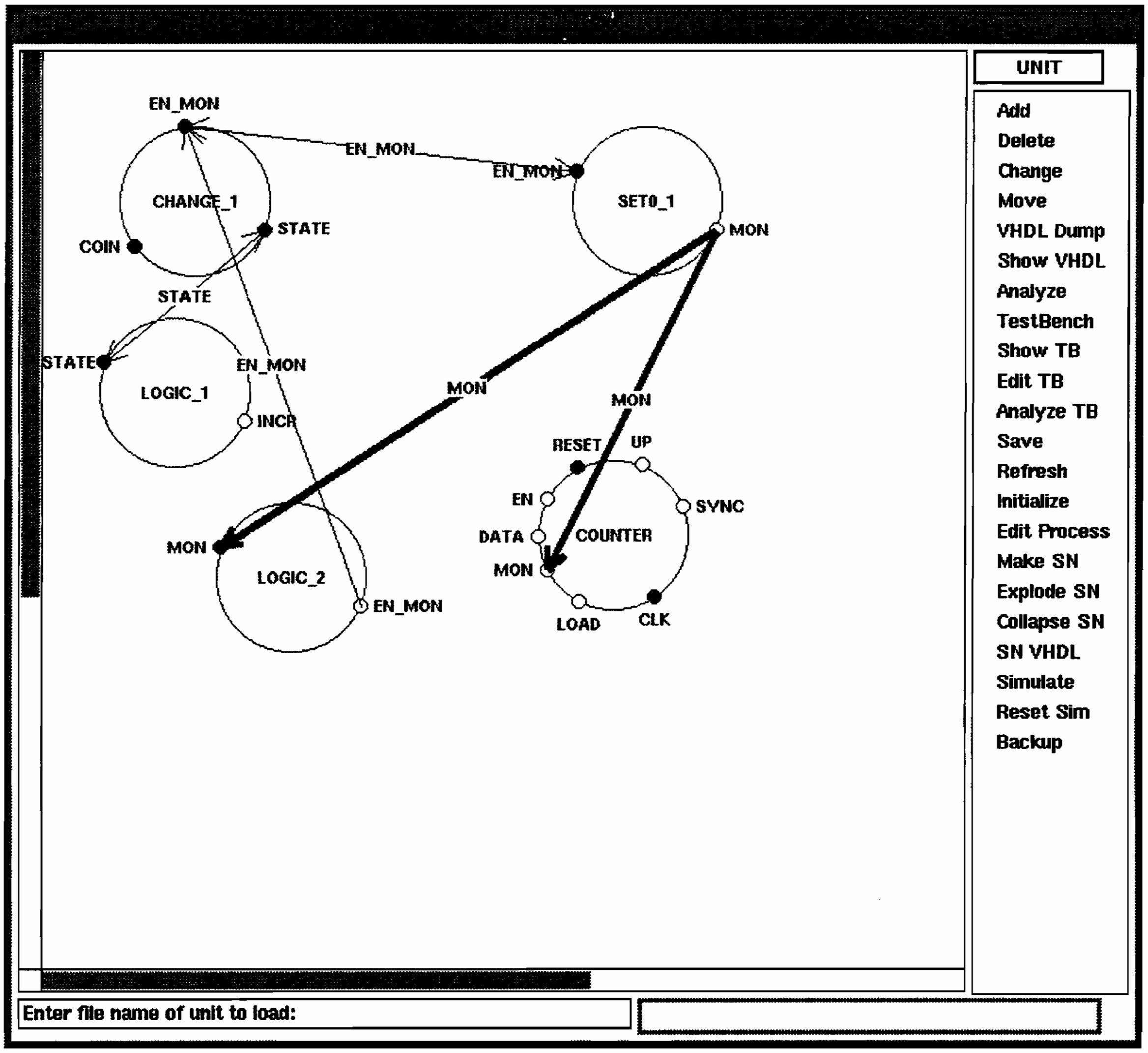
The execution window output

```
/* Result of projection */
/*  Input concepts          Schema concepts */
    ( 30 ) increment       ( 4 ) increment
    ( 31 ) MON     ( 2 ) CNT

    This concept is undefined DATA , please enter a new referent :          /* prompts to
   /                                                                        change signal
    This concept is undefined CLK , please enter a new referent :           names */
   /
    This concept is undefined EN , please enter a new referent :
   /
    This concept is undefined RESET , please enter a new referent :
   /
    This concept is undefined UP , please enter a new referent :
   /
    This concept is undefined LOAD , please enter a new referent :
   /
    This concept is undefined SYNC , please enter a new referent :
   /
    This concept is undefined CNT_DEL , please enter a new referent :
   /
    This concept is undefined CLKTRIG , please enter a new referent :
   1
    This concept is undefined ENTRIG , please enter a new referent :
   1
    This concept is undefined RESTTRIG , please enter a new referent :
   1
    This concept is undefined UPTRIG , please enter a new referent :
   1
    This concept is undefined LOADTRIG , please enter a new referent :
   1
    This concept is undefined SYNCTRIG , please enter a new referent :
   1
```

```
0 DATA DATA                                   /* updated signal
1 CLK CLK                                     list */
2 EN EN
3 RESET RESET
4 UP UP
5 LOAD LOAD
6 SYNC SYNC
7 CNT MON
8 CNT_DEL CNT_DEL
9 CLKTRIG 1
10 ENTRIG 1
11 RESTTRIG 1
12 UPTRIG 1
13 LOADTRIG 1
14 SYNCTRIG 1

/* modified VHDL code */

if (RESET = '1' and RESET'EVENT and SYNC /= '1') or
  (RESET = '1' and SYNC = '1' and CLK = '1' and
   CLK'EVENT) then
 MON <= "0000" after MON_DEL;
elsif (LOAD = '1' and CLK = '1' and CLK'EVENT) then
 MON <= DATA after MON_DEL;
elsif (EN = '1' and CLK = '1' and CLK'EVENT) then
 if UP = '1' then
  MON <= INC(MON) after MON_DEL;
 else
  MON <= DEC(MON) after MON_DEL;
 end if
end if;

/* PMG database */

;    1    COUNTER 1
;    2    SYNC    2
;    4    LOAD    2
;    5    UP      2
;    6    RESET   2
;    7    EN      2
;    8    CLK     2
;    9    DATA    2
;    10   MON     2
;    3    CNT_DEL 3
;    3        0
;    3        0
```

```
/*Summary of results */

1 : process CHANGE_1  (*STATE:inout, *COIN:in, *EN_MON:in)
2 : process LOGIC_1  (INCR:out, *STATE:in)
3 : process SET0_1  (MON:out, *EN_MON:in)
4 : process LOGIC_2  (EN_MON:out, *MON:in)
5 : process COUNTER  (SYNC:inout, LOAD:inout, UP:inout, RESET:inout, EN:inout,
 CLK:inout, DATA:inout, MON:inout)
1 : sig 0 : STATE (unknown[0])
2 : sig 1 : COIN (unknown[0])
3 : sig 2 : EN_MON (unknown[0])
4 : sig 3 : INCR (unknown[0])
5 : sig 4 : MON (BIT[0])
6 : sig 5 : SYNC (unknown[0])
7 : sig 6 : LOAD (unknown[0])
8 : sig 7 : UP (unknown[0])
9 : sig 8 : RESET (unknown[0])
10 : sig 9 : EN (unknown[0])
11 : sig 10 : CLK (unknown[0])
12 : sig 11 : DATA (unknown[0])

(D)one, (A)bort writing, edit (S)ignal, (M)erge processes
change (P)rocess name, show (C)ode

d


warning!! file 'fsm.unt' exists. overwrite? y
```

The PROCESS Model Graph.



In this PMG the only primitive is that of the counter used to count the money deposited. However the only signal getting covered is MON as it was a part of the subgraph for the increment event.

APPENDIX B
Programmer Manual

The CGVHDL program is written in C++ and a lot of time was devoted to design Class definitions and data structures to make full use of the advantages of Object Oriented Programming.

B1 The Division of Tasks with Classes

As explained in Chapter 5 the main blocks of CGVHDL are,

(i) The Input preprocessor : This uses the Taxonomy Class to setup the concept type hierarchy.

(ii) The Primitive detector : This is performed by the Primitive Graph class which is a derived class from the original Graph Class.

(iii) The Projection Engine : The entire projection algorithm is performed by the Mapper Class.

(iv) Process Mapper: A couple of functions had to be added to the existing Process Class and some had to be modified. However all modifications made were checked to ensure that previous operation was unaffected.

(v) Entity Processor: This is also invoked by the Process Class but some new functions were added to the file interfac.h.

The Taxonomy, Graph, Activity, Objects, Links, Process Classes were originally written by *[Honcharik 93]*. Most of these classes have undergone major changes to integrate the new algorithm. Also, all console and file I/O has been changed to use the C++ IoStream Class for rather than standard C routines provided in *stdio.h.*

B2 The Taxonomy Class

The header file for the class makes the following definitions,

```
struct taxtype
 {
  char        tname[16];     // type name
  int   tid;           // type id
  taxtype     *subtypes;     // list of subtypes
  taxtype     *next;         // other types at same hierarchy level

  taxtype()   { next = subtypes = NULL; tid=tname[0]=0; }
 };
```

```
struct taxonomy
 {
 taxtype        *top, *list[50];
 int            numtypes;

 void show(taxtype *a, int ind);
 inline void showall() { show(top,0); };
 void elim(taxtype *a, int el[]);
 taxtype* read_taxonomy(char *filename);
 int find_tax_type(char *type);
 int tax_search(taxtype *tptr, char *name);
 int tax_search(taxtype *tptr, int chktyp);
 int istype(int chktyp, int maintype);
 int istype(char *name, int maintype);
 int istype(int chktyp, char *type);
 int istype(char *name, char *type);
 taxonomy()    { top=NULL; numtypes=0; }
 };
```

The function call *read_taxonomy* is used to read in a taxonomy data file called taxonomy.dbs. The function *show* is used to view the type and subtypes of a particular type entry in the hierarchy. To view the entire tree, the function *showall* is invoked which recursively calls show starting from the top of the list. The function most used in this class is istype which returns -1 if the first argument is not a subtype of the second. It may call another function tax_search which searches for a type under an entry of tax_type in the hierarchy list.

Earlier it was proposed to have the identification of primitive concepts in the taxonomy by making a type in the hierarchy called device and including various device names in it as subtypes but the input may not use the word device in the specification and thus we could have a problem. Also a primitive may be implied by behavioral concepts and action concepts hence it was decided to keep a separate database for identifiers used to detect potential primitive referring concepts.

B3 The Graph Class

This class defines the core data structure of the whole CGVHDL program. It is class which sets up all the data structure to be used for storing concepts and an organized list of concepts as a conceptual graph. The graph data structures are shown below,

```
typedef struct tidtype
 {
   int gid;
   int cid;
 } targettype;

typedef struct relation
 {
   char rtype[16];
   char ref[48];
   int numtids;
   targettype tid[3];
 } reltype;

typedef struct concept
 {
   int cid;
   char ctype[16];
   char ref[48];
   int relnum;
   reltype *rel[15];
   targettype coref[3];
 } contype;

typedef struct graph
 {
   int gid;
   int connum;
   contype *con[50];
 } graphtype;
```

```
typedef struct prgraph
{
   graphtype *gr;
   char fn[48];
 } prgraphtype;

typedef struct strpair
{
  char  orig[20];
  char  cur[20];
} sigpair;
```

In the above notation,

gid - the graph id (In a system with many graphs connected by external joins,)

cid - the concept identifying id number (needed for representation of a conceptual as explained in Sowa 84.)

A combination of the two is called a target and is stored by relations to point to the related concept In this structure the graph is stored as a list concepts. There is a field in the datatype which points to the outgoing relations from the concept. These relations further have fields of type target which specify the target concept. In this way the list may be treated as an indirect linked list. The base class reads in the file *relation.dbs* to establish all the relation types being expected from the conceptual graph processor. The class is defined as follows,

```
class cgtypes
{public:
int AGNT, OBJ, NAME, AND, COND, SIZE, NUMATTRTYPES;
void read_relationships(char *filename);
int find_att_type(char *name);
};
```

```
class con_graph : public cgtypes        // conceptual graph definition
{
public:
  graphtype *graf;
  char map[80];
  void setgraf(graphtype *gr) { graf = gr;}
  inline char *name(int cnum) { return graf->con[cnum]->ref; }
  inline char *type(int cnum) { return graf->con[cnum]->ctype; }
  inline int cid(int cnum) { return graf->con[cnum]->cid; }
  int find_concept(int conid);
  int ref(int cnum, int attr);
  int atype(int cnum, int attr);
  int find_att(int cnum, int type);
  int find_att(int cnum, char *type);
  inline int find_ref(int cnum, int type)
        { return ref(cnum,find_att(cnum,type)); }
  void read_concept(FILE *inf, contype *con);
  void read_attribute(FILE *inf, reltype *rel);
  void read_graph(char *filename);
  void show_graph(FILE *outfile);
  con_graph() { graf = NULL;};
  ~con_graph();
};
```

The conceptual graph is read by using the function *read_graph* which scans each line of the input file and accordingly calls the functions *read_concept* or *read_attribute.* Apart from reading a graph, this class provides various access oriented functions like *name*, *type*, *cid, find_concept* which provide quick access to the various attributes of a concept i.e. it's name, type, cid or the index into the concept list respectively.

B3 The Primitive Graph Class

The Primitive Graph Class is derived from the above *graphcls* base class. It provides the program with the table having entries of all the primitive referring concepts. It is also used to form the final subgraph which is the projection of the schema onto the input conceptual graph. The graph to be used is set by using the function *setgraph.* The function isprim is used to detect if a concept refers to a MODAS primitive or not. Once a new graph pointer is created, various routines are called to use the projection results to instantiate the primitive subgraph. The function *showall* can be use to display the graphs to the console or redirect it to a file.

```
primg : public con_graph
{
public:

        int all[50];
        int indx;

        int used[50];

        inline char *name(int cnum) { return graf->con[cnum]->ref;}
        inline char *type(int cnum) { return graf->con[cnum]->ctype;}
        inline int cid(int cnum) { return graf->con[cnum]->cid;}

        int  isprim(char *ref, char *fname);
        void getnodes(contype *head);
        void getprenodes();
        void getpostnodes();
        void sortgraf();
        void markused(int gr_no);
        void getremains();
        graphtype *makegraf(int gr_no);
        void showall(FILE *outfile);
        void showall(FILE *outfile, graphtype *gr,int gr_no);
        primg() ;
        ~primg();
};
```

B4 The Mapper Class

This is the core execution class used in CGVHDL. The Mapper class defines all recursive functions used in the process of projection. All that is needed to instantiate a Mapper object is, a pointer to the input graph and the name of the MODAS primitive being used for projection. The Mapper class then works with two graph pointers subgr and primgr(which points to the schema and is read from a file using the read_graph routine from the Graph Class). A structure called *back_link* is used to extend the existing graph-concept list which is an indirect singly linked to an indirect doubly linked list. The structure *sigpair* defined in the header file *graph.h* is used to store the identifier names i.e. the old names from the MODAS primitive or primitive schema and the new names assigned from the specification conceptual graph or user input.

```
typedef struct back_link
 {
        int numbak;
        int bak[5];
 } back_link_type;

class mapper
{
public:

        int conmore;
        int path;
        ifstream fp;

        char *labels[10];
        char thispath[80];
        streampos curpos;
        int pindx, sindx;
        sigpair undef_sig[20];
        int tot_ref;
        back_link_type bptr[50];
        back_link_type bprg[50];

        mapper(graphtype *gr, char *fn);

        void initprimg(void);
        void getbacklinks(graphtype *gr, int gtype);
        void map(void);

        void postmap(contype *conc);
        void premap(contype *conc, contype *sconc);
        void paircon(contype *conc, contype *sconc);
        int getpair(contype *sconc, char *rarg, char *carg);
        int getprev(contype *sconc , char *rarg, char *carg, char *cref);

        int reftoind(char *conref ,int gtype);
        int reftocid(char *conref ,int gtype);
        int head(int prev);
        int getdup(int cindx);
        int getdup(int concid, int stat, int rec);

        char *strindex(char *a , char *b);
        void back(void);
        void setpos(void);
        void getpos(void);
        void newcode(void);
        void get_unmapped(void);

private:
        primg *primgr;
        primg *subgr;
        char *primfile;
};
```

Once the mapper object is instantiated, the first functions to be called are *initprimg* and *getbacklinks*. *initprimg* initializes all the graphs, i.e., primgr and subgr. *getbacklinks* is called to use the input conceptual graph and setup a data structure that allows a bi-directional access through the graph. The function called in the main routine is *map*. This

in turn calls the various recursive functions. The first function called is *postmap* which identifies the head node in the primitive graph. Once the head nodes are identified projection starts by calling the recursive function *paircon* which also calls *getpair* which in turn uses *getdup*. This sets off a projection out from the head nodes in both graphs. Then the function *premap* is called for mapping all the prenodes of the head node. *premap* now uses the functions *getprev* and *getdup*. All nodes found to match are added to a list by cid. For each entry in this list we recursively call *paircon* and *premap*. This process on completion gives a maximal projection of the schema into the input conceptual graph.
The functions *get_unmapped*, *newcode* and *strindex* are used to read the MODAS primitive and from the available VHDL code decide on the unspecified identifiers and prompts the user for new names. The function *newcode* in turn calls *strindex* and uses the structure *sigpair* to update the VHDL code to the specification.

B5 The Activity, Object, Links Classes

These are the classes involved with the previous version of the CGVHDL Linker. These are used to generate the intermediate form of representation from the input conceptual graph. For more details on these functions refer *[Honcharik 93]*.

B6 The Process Class

The process class is used to set up the data structure required to write the PMG files for MODAS. Most of this class definition is the same as in *[Honcharik 93]*. The only new addition are a couple of functions which are now used to set up a process data structure for a primitive from the .prm file, i.e., the MODAS primitive file.

```
# ifndef __PROC__
# define  __PROC__

class ptypes
{public: enum portmodes{INOUT,IN,OUT}; };
struct vhdlstring { char str[81]; vhdlstring *next; };

struct signal { namestr name; int type, size; };

struct process : public ptypes
{
typedef signal *signalptr;
namestr name, pname[10];
signalptr ports[10];
signal gen[5], var[5];
int sense[10], pmode[10], numports, numgens, numvars;
vhdlstring *firststr, *laststr;
int chrcnt;
int add_port(char *name, signalptr sigptr, int mode);
int add_gen(char *name, int type, int size);
int add_var(char *name, int type, int size);
int find_port(char *name);
int find_gen(char *name);
int find_var(char *name);
int putsense(char *name);
int putsense(int port);
vhdlstring *get_next_vhdl(vhdlstring *str);
void clear();
void appendvhdl(char *name);
void appendvhdl(char ch);
void show_head(FILE *outfile);
void show_process(FILE *outfile);
process(char *pname);
~process();
};
#define MAX 100
```

```
class signal_list
{
public:
signal siglist[MAX];
int sigtop;
inline signal* sig(int snum) { return &(siglist[snum]); }
inline char* sname(int snum) { return siglist[snum].name; }
inline int stype(int snum) { return siglist[snum].type; }
inline void stype(int snum, int type) { siglist[snum].type = type; }
inline int ssize(int snum) {return siglist[snum].size; }
inline void ssize(int snum, int size) { siglist[snum].size = size; }
int find_sig(char* name);
int add_sig(char* name, int type, int size);
int lastsig( char* name );
void show_signal(FILE *outfile, int snum);
void show_signal_list(FILE *outfile);
signal_list() { sigtop=-1; }
};

class process_list : public signal_list, public ptypes
{
public:
process *plist[50];
int topproc;
int mark;
process* proc(int pnum) {return plist[pnum];}
process* make_process(char *name);
int last_process(char *name);
process* find(char *name);
void delete_proc(int pnum);
void show_list(FILE *outfile);
void edit_sig();
void edit_name();
int merge( int p1, int p2);
void automerge();
void merge();
void show_code();
process_list() { topproc=-1; }
};


#endif
```

A new field *mark* has been added to keep track of processes generated from a MODAS primitive. The function *automerge* has been modified. The new function *modtoproc* is used to initialize the process data structure from a MODAS .prm file. The class *signal_list* is used to set-up the signal connections between the various processes in the class *process_list*. The class *process_list* also contains functions which are used for making final changes before writing the PMG to MODAS files.

APPENDIX C

Installation and Users Manual

C1 Setup Instructions

CGVHDL must be configured before use as shown below,

```
./VLINK/cgvhdl*
./VLINK/relation.dbs
./VLINK/conditio.dbs
./VLINK/actions.dbs
./VLINK/taxonomy.dbs
./VLINK/actions.lib
./VLINK/objects.lib
./VLINK/prims.dbs
```

If rest of the modas and scehma libraries are in a directory,

```
/usr/cad/modas/primlib
```

then add,

```
        setenv PRIMDIR        /usr/cad/modas/primlib
        set path = ( $path ..../VLINK)
```

to the .cshrc file.

C2 Command Line Operation

On the command line prompt invoke the linker by,

```
$ cgvhdl <graph_name>
```

for example take the complete graph for a uart call uart.g then at the command line type,

```
$ cgvhdl uart.g
```

If the file for the graph is not specified the program will display just the taxonomy hierarchy and quit with a error message.

```
14 : u
    41 : human
    40 : name
    37 : measure
         39 : time
         38 : mem_measure
    32 : event
         42 : execute
         36 : terminate
         35 : cause
         34 : call
         33 : affect
    28 : structure
         31 : act_struct
         30 : data_struct
         29 : physic_struct
    15 : state
         16 : be
    23 : action
         27 : reset
         26 : disable
```

```
        25 : enable
        17 : move
                20 : connect
                19 : write
                        21 : output_2
                18 : read
        24 : operate
13 : object
        7 : value
                12 : information
                11 : constant
                9 : data
                        10 : address
                8 : command
        0 : device
                6 : transducer
                5 : carrier
                        22 : output_3
                4 : logic_memory
                3 : memory
                2 : logic
                1 : processor
```

Fig C1 The taxonomy hierarchy

(4) shift-register	(1) shift-register
(6) OREG	(2) Q
(1) load	(3) load
(18) shift	(4) shift
(2) data	(5) data
(5) rise	(14) rise
(19) OP	(9) COUT
(17) rise	(15) rise
(3) DATA	(6) PARDAT
(7) LOAD	(8) LOAD
(13) clock	(11) clock
(26) OCLK	(12) CLK

Fig C2 The mapping results

In figure C2, the left column shows the the concept id and referent of the input graph and the right column shows the corresponding concepts mapped with in the schema. Then the program then displays the unspecified signal and generic names used in the model and prompts the user to chnge them. If the name has to be changed, at the prompt type in the new name else type in a '/' with a return key.

```
 This concept is undefined SERDAT , please enter a new referent :
/
 This concept is undefined SR , please enter a new referent :
/
 This concept is undefined SL , please enter a new referent :
/
 This concept is undefined SREG_DEL , please enter a new referent :
/
 This concept is undefined CLKTRIG , please enter a new referent :
1
 This concept is undefined LOADTRIG , please enter a new referent :
1
 This concept is undefined SRTRIG , please enter a new referent :
1
 This concept is undefined SLTRIG , please enter a new referent :
1
```

Fig C3 The User prompts for renaming unspecified names

These messages are printed out for every primitive process that is identified and instantiated. Once all the projection operations are over then the program comes to the post processor stage where it prints outs the same commands as in the previous version for further signal type and Process editing. Some examples of final stage editing follow. The user must set all data types right in this session for best results. The MODAS tool may also be used to perform some the editing except for process merging.

```
1 : process SET_1  (NINTO:out, *LOAD:in, *OCTR:in)
2 : process SET0_2  (OCLKEN:out, *OCTR:in)
3 : process SHIFTREG  (OP:inout, OREG:inout, SL:inout, SR:inout, LOAD:inout,
SERDAT:inout, OCLK:inout, DATA:inout)
4 : process OSC  (OCLK:inout, OCLKEN:inout)
5 : process COUNTER  (SYNC:inout, LOAD:inout, UP:inout, RESET:inout, EN:inout,
OCLK:inout, DATA:inout, OCTR:inout)
1 : sig 0 : LOAD (BIT[0])
2 : sig 1 : NINTO (BIT[0])
3 : sig 2 : OCTR (unknown[0])
4 : sig 3 : OCLKEN (BIT[0])
5 : sig 4 : OP (unknown[0])
6 : sig 5 : OREG (unknown[0])
7 : sig 6 : SL (unknown[0])
8 : sig 7 : SR (unknown[0])
9 : sig 8 : SERDAT (unknown[0])
10 : sig 9 : OCLK (unknown[0])
11 : sig 10 : DATA (unknown[0])
12 : sig 11 : SYNC (unknown[0])
13 : sig 12 : UP (unknown[0])
14 : sig 13 : RESET (unknown[0])
15 : sig 14 : EN (unknown[0])

(D)one, (A)bort writing, edit (S)ignal, (M)erge processes
change (P)rocess name, show (C)ode

s                                                   /* Session to change signal
                                                       type and range */

Edit which signal? 3
Editing signal 2 : sig 2 : OCTR (unknown[0])
  0 = unknown
  1 = BIT
  2 = MVL
  3 = none
  4 = BOOLEAN
  5 = INTEGER
  6 = REAL
  7 = TIME

New signal type: 1

New signal size: 8
```

```
1 : process SET_1  (NINTO:out, *LOAD:in, *OCTR:in)
2 : process SET0_2  (OCLKEN:out, *OCTR:in)
3 : process SHIFTREG  (OP:inout, OREG:inout, SL:inout, SR:inout, LOAD:inout,
SERDAT:inout, OCLK:inout, DATA:inout)
4 : process OSC  (OCLK:inout, OCLKEN:inout)
5 : process COUNTER  (SYNC:inout, LOAD:inout, UP:inout, RESET:inout, EN:inout,
OCLK:inout, DATA:inout, OCTR:inout)
1 : sig 0 : LOAD (BIT[0])
2 : sig 1 : NINTO (BIT[0])
3 : sig 2 : OCTR (BIT[8])
4 : sig 3 : OCLKEN (BIT[0])
5 : sig 4 : OP (unknown[0])
6 : sig 5 : OREG (unknown[0])
7 : sig 6 : SL (unknown[0])
8 : sig 7 : SR (unknown[0])
9 : sig 8 : SERDAT (unknown[0])
10 : sig 9 : OCLK (unknown[0])
11 : sig 10 : DATA (unknown[0])
12 : sig 11 : SYNC (unknown[0])
13 : sig 12 : UP (unknown[0])
14 : sig 13 : RESET (unknown[0])
15 : sig 14 : EN (unknown[0])

(D)one, (A)bort writing, edit (S)ignal, (M)erge processes
change (P)rocess name, show (C)ode

p                                                   /* Session to change
                                                    process name */

Change name of which process? 1
New name for process SET_1 : DATA_RDY

1 : process DATA_RDY  (NINTO:out, *LOAD:in, *OCTR:in)
2 : process SET0_2  (OCLKEN:out, *OCTR:in)
3 : process SHIFTREG  (OP:inout, OREG:inout, SL:inout, SR:inout, LOAD:inout,
SERDAT:inout, OCLK:inout, DATA:inout)
4 : process OSC  (OCLK:inout, OCLKEN:inout)
5 : process COUNTER  (SYNC:inout, LOAD:inout, UP:inout, RESET:inout, EN:inout,
OCLK:inout, DATA:inout, OCTR:inout)
1 : sig 0 : LOAD (BIT[0])
2 : sig 1 : NINTO (BIT[0])
3 : sig 2 : OCTR (BIT[8])
```

```
4 : sig 3 : OCLKEN (BIT[0])
5 : sig 4 : OP (unknown[0])
6 : sig 5 : OREG (unknown[0])
7 : sig 6 : SL (unknown[0])
8 : sig 7 : SR (unknown[0])
9 : sig 8 : SERDAT (unknown[0])
10 : sig 9 : OCLK (unknown[0])
11 : sig 10 : DATA (unknown[0])
12 : sig 11 : SYNC (unknown[0])
13 : sig 12 : UP (unknown[0])
14 : sig 13 : RESET (unknown[0])
15 : sig 14 : EN (unknown[0])

(D)one, (A)bort writing, edit (S)ignal, (M)erge processes
change (P)rocess name, show (C)ode

c                                                       /* Session to view
                                                           generated code */

Show code for which process? 1
process DATA_RDY  (NINTO:out, *LOAD:in, *OCTR:in)
if ((LOAD='1') and LOAD'event) then
NINTO <= '1';
end if;
if (OCTR=8) then
NINTO <= '0';
end if;

Hit a key and <Enter>

a

1 : process DATA_RDY  (NINTO:out, *LOAD:in, *OCTR:in)
2 : process SET0_2  (OCLKEN:out, *OCTR:in)
3 : process SHIFTREG  (OP:inout, OREG:inout, SL:inout, SR:inout, LOAD:inout,
SERDAT:inout, OCLK:inout, DATA:inout)
4 : process OSC  (OCLK:inout, OCLKEN:inout)
5 : process COUNTER  (SYNC:inout, LOAD:inout, UP:inout, RESET:inout, EN:inout,
OCLK:inout, DATA:inout, OCTR:inout)
1 : sig 0 : LOAD (BIT[0])
2 : sig 1 : NINTO (BIT[0])
3 : sig 2 : OCTR (BIT[8])
4 : sig 3 : OCLKEN (BIT[0])
```

```
5 : sig 4 : OP (unknown[0])
6 : sig 5 : OREG (unknown[0])
7 : sig 6 : SL (unknown[0])
8 : sig 7 : SR (unknown[0])
9 : sig 8 : SERDAT (unknown[0])
10 : sig 9 : OCLK (unknown[0])
11 : sig 10 : DATA (unknown[0])
12 : sig 11 : SYNC (unknown[0])
13 : sig 12 : UP (unknown[0])
14 : sig 13 : RESET (unknown[0])
15 : sig 14 : EN (unknown[0])

(D)one, (A)bort writing, edit (S)ignal, (M)erge processes
change (P)rocess name, show (C)ode

m                                                    /* Session to merge
                                                        processes */
Enter processes to merge: 1 2

1 : process DATA_RDY  (NINTO:out, *LOAD:in, *OCTR:in, OCLKEN:out)
2 : process SHIFTREG  (OP:inout, OREG:inout, SL:inout, SR:inout, LOAD:inout,
SERDAT:inout, OCLK:inout, DATA:inout)
3 : process OSC  (OCLK:inout, OCLKEN:inout)
4 : process COUNTER  (SYNC:inout, LOAD:inout, UP:inout, RESET:inout, EN:inout,
OCLK:inout, DATA:inout, OCTR:inout)
1 : sig 0 : LOAD (BIT[0])
2 : sig 1 : NINTO (BIT[0])
3 : sig 2 : OCTR (BIT[8])
4 : sig 3 : OCLKEN (BIT[0])
5 : sig 4 : OP (unknown[0])
6 : sig 5 : OREG (unknown[0])
7 : sig 6 : SL (unknown[0])
8 : sig 7 : SR (unknown[0])
9 : sig 8 : SERDAT (unknown[0])
10 : sig 9 : OCLK (unknown[0])
11 : sig 10 : DATA (unknown[0])
12 : sig 11 : SYNC (unknown[0])
13 : sig 12 : UP (unknown[0])
14 : sig 13 : RESET (unknown[0])
15 : sig 14 : EN (unknown[0])

(D)one, (A)bort writing, edit (S)ignal, (M)erge processes
change (P)rocess name, show (C)ode
```

```
c                                                                   /* view code
Show code for which process? 1                                      for the merged
                                                                    process */
process DATA_RDY  (NINTO:out, *LOAD:in, *OCTR:in)
if ((LOAD='1') and LOAD'event) then
NINTO <= '1';
end if;
if (OCTR=8) then
NINTO <= '0';
end if;
if (OCTR = "1000") then
OCLKEN <= '0'
end if;

1 : process DATA_RDY  (NINTO:out, *LOAD:in, *OCTR:in, OCLKEN:out)
2 : process SHIFTREG  (OP:inout, OREG:inout, SL:inout, SR:inout, LOAD:inout,
SERDAT:inout, OCLK:inout, DATA:inout)
3 : process OSC  (OCLK:inout, OCLKEN:inout)
4 : process COUNTER  (SYNC:inout, LOAD:inout, UP:inout, RESET:inout, EN:inout,
OCLK:inout, DATA:inout, OCTR:inout)
1 : sig 0 : LOAD (BIT[0])
2 : sig 1 : NINTO (BIT[0])
3 : sig 2 : OCTR (BIT[8])
4 : sig 3 : OCLKEN (BIT[0])
5 : sig 4 : OP (unknown[0])
6 : sig 5 : OREG (unknown[0])
7 : sig 6 : SL (unknown[0])
8 : sig 7 : SR (unknown[0])
9 : sig 8 : SERDAT (unknown[0])
10 : sig 9 : OCLK (unknown[0])
11 : sig 10 : DATA (unknown[0])
12 : sig 11 : SYNC (unknown[0])
13 : sig 12 : UP (unknown[0])
14 : sig 13 : RESET (unknown[0])
15 : sig 14 : EN (unknown[0])

(D)one, (A)bort writing, edit (S)ignal, (M)erge processes
change (P)rocess name, show (C)ode

d
```

Fig C4 A Session of Editing signal names and types, and process names within CGVHDL

APPENDIX D

Library and Database files

D1 The Taxonomy Database

```
processor isa device.
logic isa device.
memory isa device.
logic_memory isa device.
carrier isa device.
transducer isa device.
command isa value.
data isa value.
address isa data.
constant isa value.
information isa value.
device isa object.
value isa object.
object isa u.
```

```
be isa state.
read isa move.
write isa move.
connect isa move.
output_2 isa write.
output_3 isa carrier.
operate isa action.
move isa action.
enable isa action.
disable isa action.
reset isa action.
action isa u.
state isa u.
physic_struct isa structure.
data_struct isa structure.
act_struct isa structure.
structure isa u.
affect isa event.
call isa event.
cause isa event.
terminate isa event.
event isa u.
mem_measure isa measure.
time isa measure.
measure isa u.
name isa u.
human isa u.
execute isa event.
```

D2 The Relation List

gindx
agnt
obj
src
dest
det
in
inst
loc
name
val
and
or
cond
attr
duration
size
dir
freq
after

D3 The Primitive List

shift-register	SHIFTREG
shift	SHIFTREG
shifter	SHIFTREG
shift	SHIFTREG
shifted	SHIFTREG
clock	OSC
counter	COUNTER
increment	COUNTER
decrement	COUNTER
register	REGISTER
load	REGISTER
write	REGISTER
buffer	BUFFER

D4 The Schema Library

Some of the commomnly used schemas are as follows,

1. A Shift Register

```
[ 1 : device : shift-register ]
        ->( name : null ) -> [ 2 ]
[ 2 : id : Q ]
[ 3 : write : load ]
        ->( gindx : 1 )
        ->( obj : null ) -> [ 5 ]
        ->( dest : into ) -> [ 1 ]
        ->( cond : when ) -> [ 7 ]
[ 4 : operate : shift ]
        ->( obj : null ) -> [ 2 ]
        ->( dest : in ) -> [ 9 ]
        ->( cond : when ) -> [ 10 ]
[ 5 : data : data ]
        ->( name : null ) -> [ 6 ]
[ 6 : id : PARDAT ]
[ 14 : event : rise ]
        ->( agnt : null ) -> [ 8 ]
```

```
[  8 : id : LOAD ]
[  9 : id : COUT ]
[ 15 : event : rise ]
        ->( agnt : null )  -> [ 11 ]
[ 11 : device : clock ]
        ->( name : null )  -> [ 12 ]
[ 12 : id : CLK ]
[ 7 :state : is ]
        ->( agnt : null ) -> [ 8 ]
        ->( attr : null ) -> [ 13 ]
[ 10 :state : is ]
        ->( agnt : null )  -> [ 11 ]
        ->( attr : null ) -> [ 13 ]
[ 13 : constant : 1 ]
#1,3,4(7:14)(10:15)(14:7)(15:10)
```

2. A Register

```
[  1 : device : register ]
          ->( name : null )  -> [ 2 ]
[  2 : id : Q ]
[  3 : event : reset ]
          ->( obj : null )  -> [ 1 ]
          ->( attr : null )  -> [ 6 ]
          ->( cond : when )  -> [ 13 ]
[  5 : state : is ]
          ->( agnt : null )  -> [ 1 ]
          ->( attr : null )  -> [ 12 ]
[  6 : constant : 0 ]
[  7 : event : rise ]
          ->( agnt : null )  -> [ 8 ]
[  8 : id : RESET ]
[  9 : event : rise ]
          ->( agnt : null )  -> [ 10 ]
[ 10 : device : clock ]
          ->( name : null )  -> [ 11 ]
[ 11 : id : CLK ]
[ 12 : constant : 8 ]
[ 13 : u : or ]
          ->( or : null ) -> [ 14 ]
          ->( or : null ) -> [ 19 ]
[ 14 : u : and ]
```

```
          ->( and : null ) -> [ 7 ]
          ->( and : null ) -> [ 15 ]
[ 15 : state : is not ]
          ->( agnt : null ) -> [ 16 ]
          ->( attr : null ) -> [ 17 ]
[ 16 : id : SYNC ]
[ 17 : constant : 1 ]
[ 18 : state : is ]
          ->( agnt : null ) -> [ 16 ]
          ->( attr : null ) -> [ 6 ]
[ 19 : u : and ]
          ->( and : null ) -> [ 20 ]
          ->( and : null ) -> [ 21 ]
          ->( and : null ) -> [ 9 ]
[ 20 : state : is ]
          ->( agnt : null )  -> [ 8 ]
          ->( attr : null ) -> [ 17 ]
[ 21 : state : is ]
          ->( agnt : null )  -> [ 16 ]
          ->( attr : null ) -> [ 17 ]
[ 22 : action : increment ]
          ->( obj : null ) -> [ 1 ]
          ->( cond : null ) -> [ 23 ]
[ 23 : u : and ]
          ->( and : null ) -> [ 9 ]
          ->( and : null ) -> [ 24 ]
          ->( and : null ) -> [ 25 ]
[ 24 : state : is ]
          ->( agnt : null) -> [ 26 ]
          ->( attr : null ) -> [ 17 ]
[ 25 : state : is ]
          ->( agnt : null) -> [ 27 ]
          ->( attr : null ) -> [ 17 ]
[ 26 : id : EN ]
[ 27 : id : UP ]
[ 28 : action : decrement ]
          ->( obj : null ) -> [ 1 ]
          ->( cond : null ) -> [ 29 ]
[ 29 : u : and ]
          ->( and : null ) -> [ 9 ]
          ->( and : null ) -> [ 24 ]
          ->( and : null ) -> [ 30 ]
[ 30 : state : is ]
```

```
            ->( agnt : null) -> [ 27 ]
            ->( attr : null ) -> [ 6 ]
#1,3,4(15:18)(18:15)
```

3. A Clock or Oscillator

```
[ 1 : device : clock ]
            ->( name : null) -> [ 2 ]
[ 2 : id : CLK ]
[ 3 : action : enable ]
            ->( obj : null )  -> [  1  ]
            ->( cond : when )  -> [  5  ]
[ 4 : apply : to ]
            ->( carrier : null ) -> [ 2 ]
            ->( attr : null ) -> [ 7 ]
            ->( swf : null ) -> [ 8 ]
            ->( cond : when ) -> [ 5 ]
            ->( duration : null ) -> [ 9 ]
[ 5 : state : is ]
            ->( agnt : null )  -> [  6  ]
            ->( attr : null ) -> [ 12 ]
[ 6 : id : CTRL ]
[ 7 : id : 1 ]
[ 8 : apply : to ]
            ->( carrier : null ) -> [ 2 ]
            ->( attr : null ) -> [ 10 ]
            ->( swf : null ) -> [ 4 ]
            ->( cond : when ) -> [ 5 ]
            ->( duration : null ) -> [ 11 ]
[ 9 : id : HITIME ]
[10 : id : 0 ]
[11 : id : LOTIME ]
[12 : id : CTRLTRIG]
[13 : event : rise]
            ->(agnt : null ) -> [ 6 ]
#1,4,8,3(5:13)(13:5)
```

4. A Counter

```
[ 1 : device : register ]
            ->( name : null )  -> [ 2 ]
```

```
[  2 : id : Q ]
[  3 : event : reset ]
                ->( obj : null )  -> [ 1 ]
                ->( attr : null )  -> [ 6 ]
                ->( cond : when )  -> [ 13 ]
[  5 : state : is ]
                ->( agnt : null )  -> [ 1 ]
                ->( attr : null )  -> [ 12 ]
[  6 : constant : 0 ]
[  7 : event : rise ]
                ->( agnt : null )  -> [ 8 ]
[  8 : id : RESET ]
[  9 : event : rise ]
                ->( agnt : null )  -> [ 10 ]
[ 10 : device : clock ]
                ->( name : null )  -> [ 11 ]
[ 11 : id : CLK ]
[ 12 : constant : 8 ]
[ 13 : u : or ]
                ->( or : null ) -> [ 14 ]
                ->( or : null ) -> [ 19 ]
[ 14 : u : and ]
                ->( and : null ) -> [ 7 ]
                ->( and : null ) -> [ 15 ]
[ 15 : state : is not ]
                ->( agnt : null ) -> [ 16 ]
                ->( attr : null ) -> [ 17 ]
[ 16 : id : SYNC ]
[ 17 : constant : 1 ]
[ 18 : state : is ]
                ->( agnt : null ) -> [ 16 ]
                ->( attr : null ) -> [ 6 ]
[ 19 : u : and ]
                ->( and : null ) -> [ 20 ]
                ->( and : null ) -> [ 21 ]
                ->( and : null ) -> [ 9 ]
[ 20 : state : is ]
                ->( agnt : null )  -> [ 8 ]
                ->( attr : null ) -> [ 17 ]
[ 21 : state : is ]
                ->( agnt : null )  -> [ 16 ]
                ->( attr : null ) -> [ 17 ]
[ 22 : action : increment ]
```

```
            ->( obj : null ) -> [ 1 ]
            ->( cond : null ) -> [ 23 ]
[ 23 : u : and ]
            ->( and : null ) -> [ 9 ]
            ->( and : null ) -> [ 24 ]
            ->( and : null ) -> [ 25 ]
[ 24 : state : is ]
            ->( agnt : null) -> [ 26 ]
            ->( attr : null ) -> [ 17 ]
[ 25 : state : is ]
            ->( agnt : null) -> [ 27 ]
            ->( attr : null ) -> [ 17 ]
[ 26 : id : EN ]
[ 27 : id : UP ]
[ 28 : action : decrement ]
            ->( obj : null ) -> [ 1 ]
            ->( cond : null ) -> [ 29 ]
[ 29 : u : and ]
            ->( and : null ) -> [ 9 ]
            ->( and : null ) -> [ 24 ]
            ->( and : null ) -> [ 30 ]
[ 30 : state : is ]
            ->( agnt : null) -> [ 27 ]
            ->( attr : null ) -> [ 6 ]
#1,3,4(15:18)(18:15)
```

D5 The Actions Library *[Honcharik 93]*

```
apply(agnt=[event,state],obj,dest)=zbuffer1.
apply(cond,obj,dest)=zbuffer1.
apply(obj,dest)= connect.
buffer(agnt=[event,state],obj,dest)=zbuffer1.
buffer(cond,obj,dest)=zbuffer1.
change(dest,attr)=change1.
change(obj)=change2.
connect(cond,obj,dest)=zbuffer1.
connect(obj,dest)= connect.
disable(obj=device)=disable.
enable(obj=device)=enable.
equal(cond,agnt,attr)=logic.
equal(agnt,attr)=is.
```

```
execute(obj)=execute.
fall(agnt)=fall.
follow(obj,dest)=connect.
increment(obj)=increment.
interrupt(obj=device,inst)=interrupt.
invert(obj,freq:every)=invert1.
is(cond,agnt,attr)=logic.
is(agnt,attr)=is.
keep(dest,attr)=keep.
latch(obj,dest)=connect.
load(gindx:1)= load1.
load(obj,dest)=load1.
load(obj=memory, src=[data,constant,address])= load2.
load(obj,src)=load2.
output(dest,obj)= connect.
read(obj,src,dest)=load3.
read(src,dest)=memread.
reset(obj)=reset2.
reset()=reset1.
rise(agnt)=rise.
set(obj,after)=set4.
set(obj,attr)=set3.
set(obj,duration)=set2.
set(obj)=set1.
shift(obj,dir=value:left,src,dest)= shift_lft.
shift(obj,dir=value:right,src,dest)= shift_rt.
shift(obj,dir=value:left,dest)= shift_lft0.
shift(obj,dir=value:right,dest)= shift_rt0.
shift(obj,dir=value:left,src)= shift_lft1.
shift(obj,dir=value:right,src)= shift_rt1.
store(src,dest)=memstore.
transfer(obj,src,dest)=load3.
```

D6 The Objects Library *[Honcharik 93]*

```
address:address(name) { type=signal; name=name; value="QWER";}
constant:high() { type=value; name="high"; value="'1'"; stype=BIT; size=1;}
constant:low() { type=value; name="low"; value="'0'"; stype=BIT; size=1;}
data:data(loc=address(name)) {newobj=loc; value="ABCD"; stype=BIT;}
data:data(size) { type=signal; name="data"; value="data"; stype=BIT; size=size;}
```

```
memory:register(name=[id,name]) { type=signal; name=name; stype=BIT;}
output_3:output(agnt=device) { newobj=agnt; }
device:clock() { type=signal; name="clock"; stype=BIT; size=1;}
pin:pin() {type=signal; stype=BIT; size=1;}
instruction:instruction(name) { type=var/value; name="instruction"; value=name;}
input_2:input(agnt=device) { type=signal; name="datain"; }
value() { type=value; name=ref; value=ref; }
```

D7 The Actions Database *[Honcharik 93]*

```
apply
{
*APPLY(*obj:in,dest:out)
{[dest] <= [obj];
}
}
change1
{
*CHANGE(dest:out)
{[dest] <= [attr];
}
}
connect
{
*CONN(*obj:in=dest:out)
{[dest] <= [obj];
}
}
disable
{
*DISABLE(obj:out:BIT)
{[obj] <= '0';}
}
enable
{
*ENABLE(obj:out:BIT)
{[obj] <= '1';
}
}
increment
{
*INCREM(obj:inout)
{[obj] <= INC([obj]);
}
}
interrupt
{
*INTERRUPT(inst:out:BIT)
{[inst] <= '1';
else [inst] <= '0';
```

```
}
}
invert1
{
*OSC(*obj:inout:BIT)
{[obj] <= not [obj] after [freq];
}
}
is
{
*IS(agnt:out)
{[agnt] <= [attr];
}
}
keep
{
*KEEP()
{null;
}
}
load1
{
*LOAD(obj:in=dest:out)
{[dest] <= [obj];
}
}
load2
{
*LOAD(obj:out=src:in)
{[obj] <= [src]
}
}
load3
{
*LOAD(src:in=dest:out)
{[dest] <= [src];
}
}
logic
{
*LOGIC(agnt:out)
{  [agnt] <= [attr];
else
```

```
 [agnt] <= not [attr];
}
}
memread
{
*MEMRD(src:in,dest:out)
{[dest] <= MEM(INTVAL([src]));
}
}
memstore
{
*MEMSTR(src:in,dest:in)
{MEM(INTVAL([dest])) := [src];
}
}
output
{
*OUT(dest:out=*obj:in)
{[dest] <= [obj];
}
}
reset1
{
*RESET()
{RESET <= '1';
}
}
reset2
{
*SET0(obj:out:BIT)
{[obj] <= '0';
}
}
set1
{
*SET(obj:out:BIT)
{[obj] <= '1';
}
}
set2
{
*SET(obj:out:BIT)
{[obj] <= '1', '0' after [duration];
```

```
}
}
set3
{
*SET(obj:out:BIT)
{[obj] <= [attr];
}
}
set4
{
*SET(obj:out:BIT)
{[obj] <= '1' after [after];
}
}
shift_lft
{
*SHIFT_LT(obj:inout:BIT,src:in:BIT,dest:out:BIT)
v(I:integer)
{[dest] <= [obj]([obj]'high);
for I in [obj]'high downto [obj]'low+1 loop
  [obj](I) <= [obj](I-1);
end loop;
[obj]([obj]'low) <= [src];
}
}
shift_rt
{
*SHIFT_RT(obj:inout:BIT,src:in:BIT,dest:out:BIT)
v(I:integer)
{[dest] <= [obj]([obj]'low);
for I in [obj]'low to [obj]'high-1 loop
  [obj](I) <= [obj](I+1);
end loop;
[obj]([obj]'high) <= [src];
}
}
shift_lft0
{
*SHIFT_LT(obj:inout:BIT,dest:out:BIT)
v(I:integer)
{[dest] <= [obj]([obj]'high);
for I in [obj]'high downto [obj]'low+1 loop
  [obj](I) <= [obj](I-1);
```

```
end loop;
[obj]([obj]'low) <= '0';
}
}
shift_rt0
{
*SHIFT_RT(obj:inout:BIT,dest:out:BIT)
v(I:integer)
{[dest] <= [obj]([obj]'low);
for I in [obj]'low to [obj]'high-1 loop
  [obj](I) <= [obj](I+1);
end loop;
[obj]([obj]'high) <= '0';
}
}
shift_lft1
{
*SHIFT_LT(obj:inout:BIT,src:in:BIT)
v(I:integer)
{for I in [obj]'high downto [obj]'low+1 loop
  [obj](I) <= [obj](I-1);
end loop;
[obj]([obj]'low) <= [src];
}
}
shift_rt1
{
*SHIFT_RT(obj:inout:BIT,src:in:BIT)
v(I:integer)
{for I in [obj]'low to [obj]'high-1 loop
  [obj](I) <= [obj](I+1);
end loop;
[obj]([obj]'high) <= [src];
}
}
tristate
{
*TRI(obj:out:MVL)
v(I:integer)
{for I in [obj]'range loop
 [obj](I) <= 'Z';
end loop;
}
```

```
}
zbuffer1
{
*ZBUF(*obj:in:BIT=dest:out:MVL)
v(I:integer)
{[dest] <= BV_TO_MVL([obj]);
else
 for I in [dest]'range loop
   [dest](I) <= 'Z';
 end loop;
}
}
```

D8 The Conditions Database *[Honcharik 93]*

```
change2
{*(*obj:in){[obj]'event}
}
enable
{*(*obj:in:BIT){[obj]=1}
}
execute
{
EXECUTE(*obj:in,imed:out:BIT)
{if [obj]=[objval] then
 [imed] <= 1;
end if;
}
*(*imed){[imed]=1}
}
fall
{
*(*agnt:in:BIT){([agnt]='0') and [agnt]'event}
}
is
{
*(*agnt:in){[agnt]=[attr]}
}
reset
{
*(){RESET='1'}
}
rise
{
*(*agnt:in:BIT){([agnt]='1') and [agnt]'event}
}
```

VITA

Vikram Shrivastava was born on January 9, 1970 in Bombay, India. He graduated from Jamnabai Narsee School, Bombay, India, in May 1985, and from Mithibai College, Bombay, India in June 1987. He attended Fr. Conceicao Rodrigues College of Engineering, University of Bombay, Bombay, India, and obtained his Bachelor of Engineering degree in Electronics *(Honors)* in July 1991. He was admitted to graduate school at the Virginia Polytechnic Institute and State University in January 1992 and received a Master of Science degree in Electrical Engineering in June 1994.

[illegible]