Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

Z
699
C3
no. 90-06

Translating BIF into VHDL: Algorithms and Examples

Tedd Hadley
Joong Hwee Cho
Nikil Dutt

Technical Report #90-06
July 12, 1990

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
(714) 856-7063

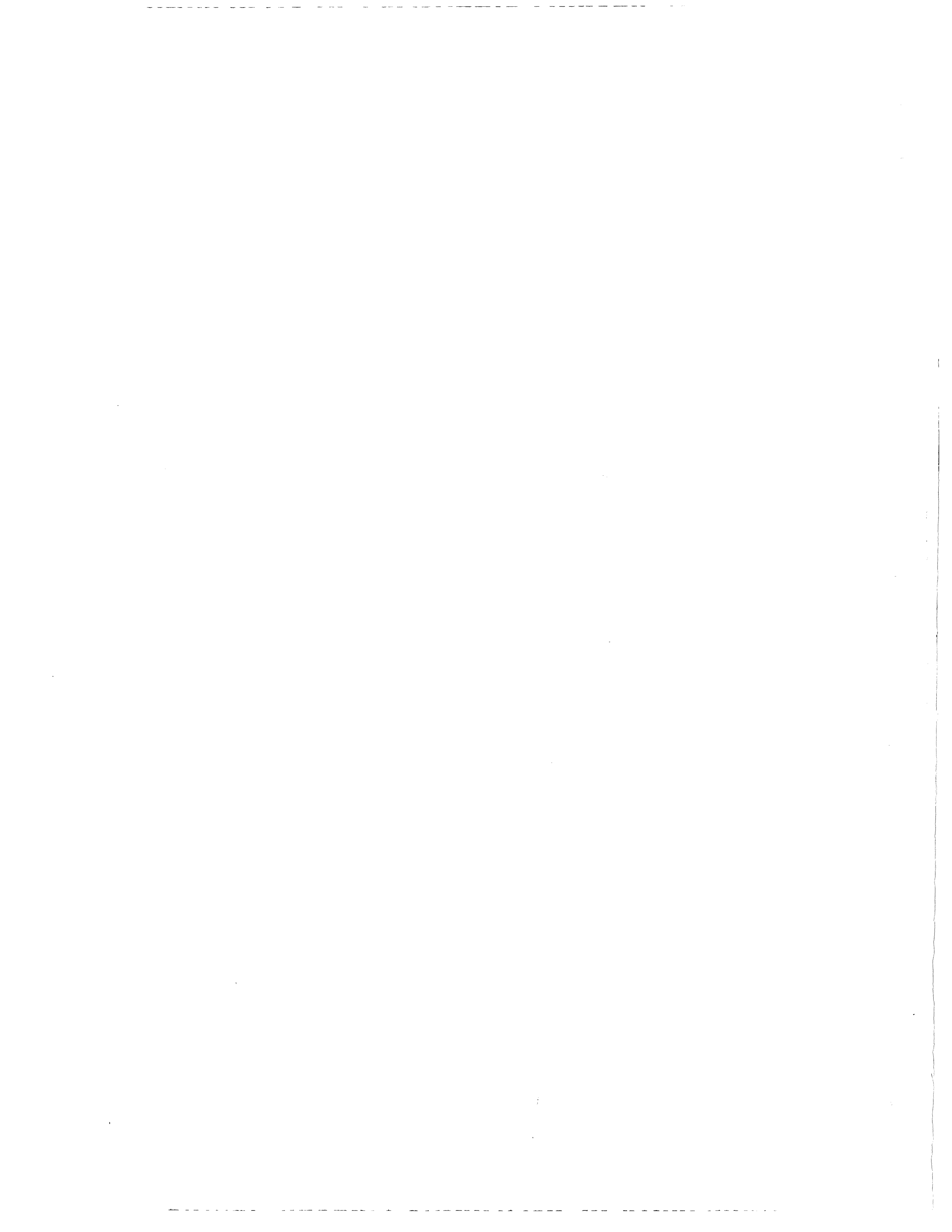
hadley@ics.uci.edu
jcho@ics.uci.edu
dutt@ics.uci.edu

Abstract

This report describes an algorithm for automatically translating BIF system-level behavioral descriptions to behavioral VHDL. BIF is a new intermediate representation for behavioral synthesis, based on annotated state tables that supports user control of the synthesis process by allowing specification of partial design structures, unit bindings, and modification of the design at various levels of abstraction. This flexibility creates a need for behavioral verification of the design at each level of abstraction to provide feedback information to the user. Since VHDL is a well formalized, simulatable language it makes an ideal target for translation.

We discuss the complexities inherent in representing BIF's hierarchical state specifications in VHDL and examine a general model for the combined representation of hierarchy, timing, concurrency, and arbitrary state transitions in VHDL.

We conclude the report with several examples from a recently implemented translator.



Contents

1 Introduction 1

2 Behavioral Modeling for Synthesis 3

3 Issues in Translating BIF Into VHDL 7

3.1 Representation of Behavioral Hierarchy and Concurrency 7

3.2 Hierarchical Events and Time-outs 9

3.3 The Table-Tree 9

3.4 Zero-Trees 12

3.5 Resolving Hierarchical Events in VHDL 13

3.5.1 Wait Statements 13

3.5.2 State Signals and Resolution Functions 15

3.6 VHDL Code Templates 17

3.7 Auxiliary VHDL Signals 19

3.8 Overall Structure of Translated VHDL Code 22

4 The Translation Algorithm 23

4.1 Outline of Translation Algorithm 23

5 A Detailed Example 26

6 Experiments 35

7 Summary 36

8 Acknowledgements 37

9 References 38

A Appendix 39

A.1 Technical Report Example: BIF 39

A.2 Technical Report Example: VHDL 43

A.3 Supplementary Example 1: BIF 49

A.4 Supplementary Example 1: VHDL 51

A.5 Supplementary Example 2: BIF 54

A.6 Supplementary Example 2: VHDL 56

A.7 Supplementary Example 3: BIF 59

A.8 Supplementary Example 3: VHDL 62

List of Figures

1	BIF Design Environment	5
2	BIF Example	8
3	Edge Weights	10
4	BIF Table-Tree	11
5	Edge Weight 1	11
6	Edge Weight 2	12
7	Edge Weight 0	13
8	Reparenting a Zero-Tree	13
9	Hierarchy Using Wait Statements	14
10	Events and Time-outs In Wait Statements	15
11	Signal Resolution	16
12	An Example Using the Block Template	18
13	The Block Template with Time-outs	18
14	An Example Using the Block-Process Template	20
15	The Use of the *bif_case* Variable	21
16	Organization of Translated VHDL Code.	22
17	Experiments	35

1 Introduction

BIF is an intermediate design representation format, based on annotated textual state tables, that captures the complete behavior and structure of a design at each level of the behavioral synthesis process. (The format is described in detail in [DuHG89] and [DuHG90].) This report presents a method for translating BIF behavioral descriptions into VHDL. It seems necessary then, before we describe the details of the algorithm, to justify the need for BIF as a behavioral representation, rather than VHDL. Why do we use BIF to describe the behavior and structure of a design initially and not VHDL? Several reasons follow.

VHDL is primarily a *simulation language* and thus its semantics are closely based on that of an event-driven simulator with the implicit notion of simulation time. Unfortunately, these semantics make the task of synthesis harder. There is no direct correspondence to hardware for features such as resolution functions, which are present in the language primarily to support simulation. Furthermore, since behavioral descriptions are written with the implicit notion of a simulation clock, several constructs in the language (such as events indicating signal changes) do not have feasible or efficient realizations in hardware. Designers also find it hard to mix styles when trying to describe designs for both synthesis and simulation using VHDL. BIF avoids these problems because it is designed to be a language primarily for synthesis. The translation algorithm, presented later in this report, makes the simulation of BIF unnecessary by translating directly to simulatable VHDL.

VHDL combines behavioral, timing, and structural models into one language. Each model has its own particular semantics, features, and syntax. From the designer's perspective, this flexibility makes VHDL a highly complex and confusing medium, both conceptually and syntactically. Thus, the learning curve for designers is low, and most designers who are used to a single model for design description find VHDL difficult to use. Furthermore, the richness of the language allows different designers to describe the same behavior in many dissimilar ways. Although these descriptions may simulate correctly, they create problems in synthesis, since synthesis tools have to account for a myriad of different description styles. BIF, on the other hand, uses a single model with a small but adequate set of common description features and integrates multiple levels of abstraction into

a concept with which designers are immediately familiar and comfortable: state tables. BIF also maintains a comparable syntax across these levels, reducing the amount of time and effort expended by the designer to learn and use the format.

As a final reason, we will show in the next sections that certain desirable features for description are not explicitly supported by VHDL. Specifically, behavioral hierarchy, time-outs, and hierarchical events, though extensively used in specifying the behavior of complex designs, do not have obvious implementations in VHDL.

In the next section we discuss some of the issues relevant to behavioral modeling and how BIF addresses them. Emerging from this discussion is the need for simulation of the design representation throughout the synthesis process and thus the need for translation from BIF to simulatable VHDL. Section 3 describes the issues involved in the translation process, section 4 lists the steps of the translation algorithm, and section 5 exhaustively details a translation example. Section 6 presents a table of designs that have been translated, and section 7 summarizes the report. Further detailed examples are included in the appendices.

2 Behavioral Modeling for Synthesis

Behavior can be defined as the functional interpretation of a design. Behavioral modeling for synthesis provides a method for describing the design's function so that it may subsequently be realized in hardware. The behavioral description can then be successively refined in phases by various synthesis tools or mechanisms which gradually map behavior to structure until the entire description is transformed to the desired hardware representation. Synthesis can be targeted to register transfer descriptions, logic level, standard cell, and ultimately, layout description; but in every intervening phase, the intermediate structure and behavior must fulfill the initial behavioral requirement. Correctness of the description is best determined by simulating the behavior at each level of abstraction. Therefore, simulation, or a simple path to simulation, is a vital requirement for a behavioral synthesis/modeling environment.

Behavioral description, by definition, does not necessarily define any specific structure. This should not require, however, that the description be entirely free of structural information. In many cases, the user may, to some degree, be familiar with the desired structure of the design as well as having a complete knowledge of the desired behavior. An ideal design description language should allow the designer freedom to define *or mix* both the behavior and structure of a design in the same language. BIF attempts to conform to this ideal by allowing the user to specify partial design structures as initial constraints. In addition, behavior to structure bindings are permitted by allowing the user to selectively bind behavioral operators to components, and behavioral variables to storage elements or wires. For example, if the designer knows that the operator + in the behavioral description $a \leftarrow a + b$ is in a particularly time-critical portion or state of the design, then he or she may wish to specify that operator be bound to a fast adder, such as a carry-look-ahead adder, rather than a slower default: $a \leftarrow a + \{cla_adder\}\ b$.

For the remainder of this section, we divide behavioral modeling issues into two categories: functional issues relating to the behavioral language itself, such as the use of one particular modeling construct over another, and issues concerning how the language interacts with the user's environment, such as the use of graphical abstractions and/or simulation feedback.

One common modeling construct used for large designs is hierarchical representation. In BIF this concept specifically addresses the case for which behavior is most naturally described by a hierarchy of behavioral "modules" descending from a top level table of super-states to many operation "leaf" states. Hierarchy in this fashion easily applies to state tables by expanding the definition of an individual state to define or encompass a table at a lower level of hierarchy.

Concurrency, the concept of simultaneous processes, goes hand-in-hand with hierarchy for describing large, complex designs. Concurrency is also easily described in BIF by allowing the user the option to define a table as operating in parallel with another table.

Representation of asynchronous behavior is important for designs that are not clocked or that exhibit a mixture of synchronous and asynchronous behavior. With the addition of an *event* field to the general state table archetype, BIF allows transfer of control from one state to another on the basis of signal transitions.

Timing specification allows the designer to specify performance, introduce technological constraints, or simply guarantee correct behavior. BIF allows duration constraints to be placed on operations or states, the latter being represented by a *timeout* event occuring after a specified amount of time.

Figure 1 shows the BIF design environment. The intermediate format is shown in the center, synthesis tools are shown on the left, and the user interface is shown on the right. At each level of abstraction the user can, either directly or through the use of tools, update, modify, or refine the design representation. This makes BIF very suitable as an intermediate format for design capture and design exchange between high level synthesis tools. In addition, each level can be simulated to show the user exactly how the design is behaving. If errors are detected, the user can modify the description appropriately or, as in the case of synthesis tool development, modify the tool. This provides a complete feedback loop at every phase of the synthesis process, allowing the user to view the format, observe the waveform simulation, and make any or all changes needed.

The user interface to BIF, XBIF, is a graphical/tabular editor which uses labeled forms to capture the various fields of a BIF description. XBIF's textual forms have runtime syntax checking

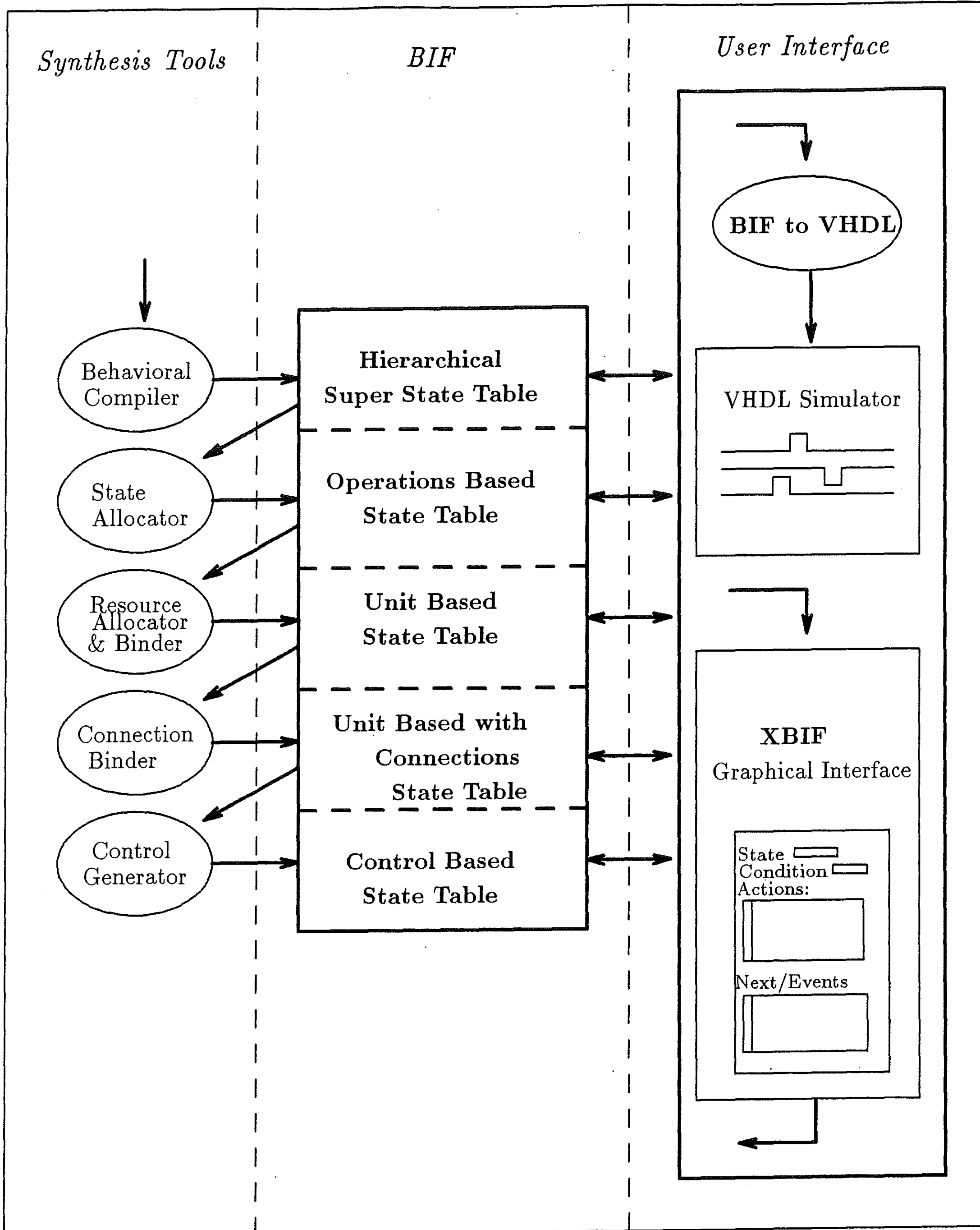


Figure 1: BIF Design Environment

One common modeling construct used for large designs is hierarchical representation. In BIF this concept specifically addresses the case for which behavior is most naturally described by a hierarchy of behavioral "modules" descending from a top level table of super-states to many operation "leaf" states. Hierarchy in this fashion easily applies to state tables by expanding the definition of an individual state to define or encompass a table at a lower level of hierarchy.

Concurrency, the concept of simultaneous processes, goes hand-in-hand with hierarchy for describing large, complex designs. Concurrency is also easily described in BIF by allowing the user the option to define a table as operating in parallel with another table.

Representation of asynchronous behavior is important for designs that are not clocked or that exhibit a mixture of synchronous and asynchronous behavior. With the addition of an *event* field to the general state table archetype, BIF allows transfer of control from one state to another on the basis of signal transitions.

Timing specification allows the designer to specify performance, introduce technological constraints, or simply guarantee correct behavior. BIF allows duration constraints to be placed on operations or states, the latter being represented by a *timeout* event occuring after a specified amount of time.

Figure 1 shows the BIF design environment. The intermediate format is shown in the center, synthesis tools are shown on the left, and the user interface is shown on the right. At each level of abstraction the user can, either directly or through the use of tools, update, modify, or refine the design representation. This makes BIF very suitable as an intermediate format for design capture and design exchange between high level synthesis tools. In addition, each level can be simulated to show the user exactly how the design is behaving. If errors are detected, the user can modify the description appropriately or, as in the case of synthesis tool development, modify the tool. This provides a complete feedback loop at every phase of the synthesis process, allowing the user to view the format, observe the waveform simulation, and make any or all changes needed.

The user interface to BIF, XBIF, is a graphical/tabular editor which uses labeled forms to capture the various fields of a BIF description. XBIF's textual forms have runtime syntax checking

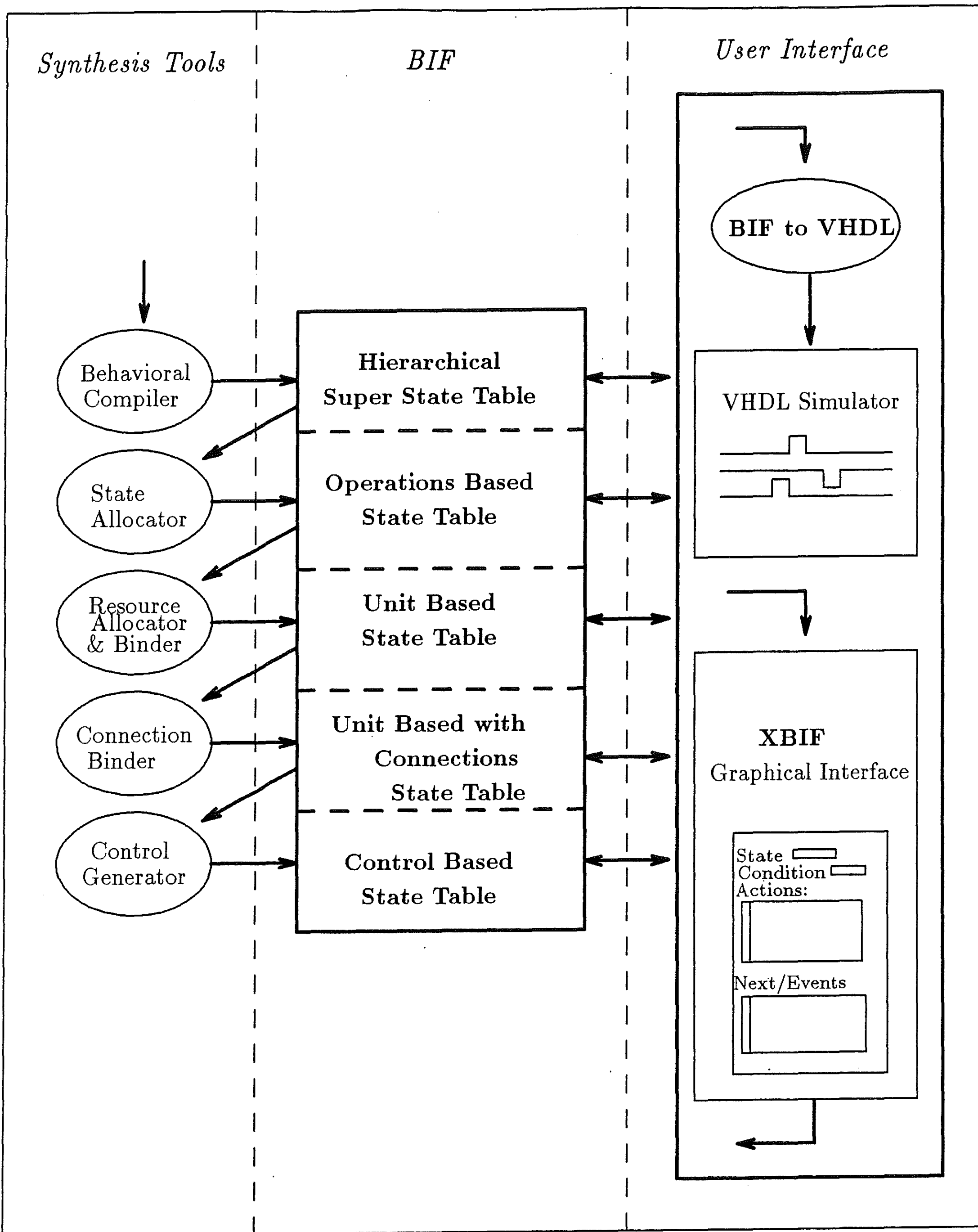


Figure 1: BIF Design Environment

One common modeling construct used for large designs is hierarchical representation. In BIF this concept specifically addresses the case for which behavior is most naturally described by a hierarchy of behavioral "modules" descending from a top level table of super-states to many operation "leaf" states. Hierarchy in this fashion easily applies to state tables by expanding the definition of an individual state to define or encompass a table at a lower level of hierarchy.

Concurrency, the concept of simultaneous processes, goes hand-in-hand with hierarchy for describing large, complex designs. Concurrency is also easily described in BIF by allowing the user the option to define a table as operating in parallel with another table.

Representation of asynchronous behavior is important for designs that are not clocked or that exhibit a mixture of synchronous and asynchronous behavior. With the addition of an *event* field to the general state table archetype, BIF allows transfer of control from one state to another on the basis of signal transitions.

Timing specification allows the designer to specify performance, introduce technological constraints, or simply guarantee correct behavior. BIF allows duration constraints to be placed on operations or states, the latter being represented by a *timeout* event occuring after a specified amount of time.

Figure 1 shows the BIF design environment. The intermediate format is shown in the center, synthesis tools are shown on the left, and the user interface is shown on the right. At each level of abstraction the user can, either directly or through the use of tools, update, modify, or refine the design representation. This makes BIF very suitable as an intermediate format for design capture and design exchange between high level synthesis tools. In addition, each level can be simulated to show the user exactly how the design is behaving. If errors are detected, the user can modify the description appropriately or, as in the case of synthesis tool development, modify the tool. This provides a complete feedback loop at every phase of the synthesis process, allowing the user to view the format, observe the waveform simulation, and make any or all changes needed.

The user interface to BIF, XBIF, is a graphical/tabular editor which uses labeled forms to capture the various fields of a BIF description. XBIF's textual forms have runtime syntax checking

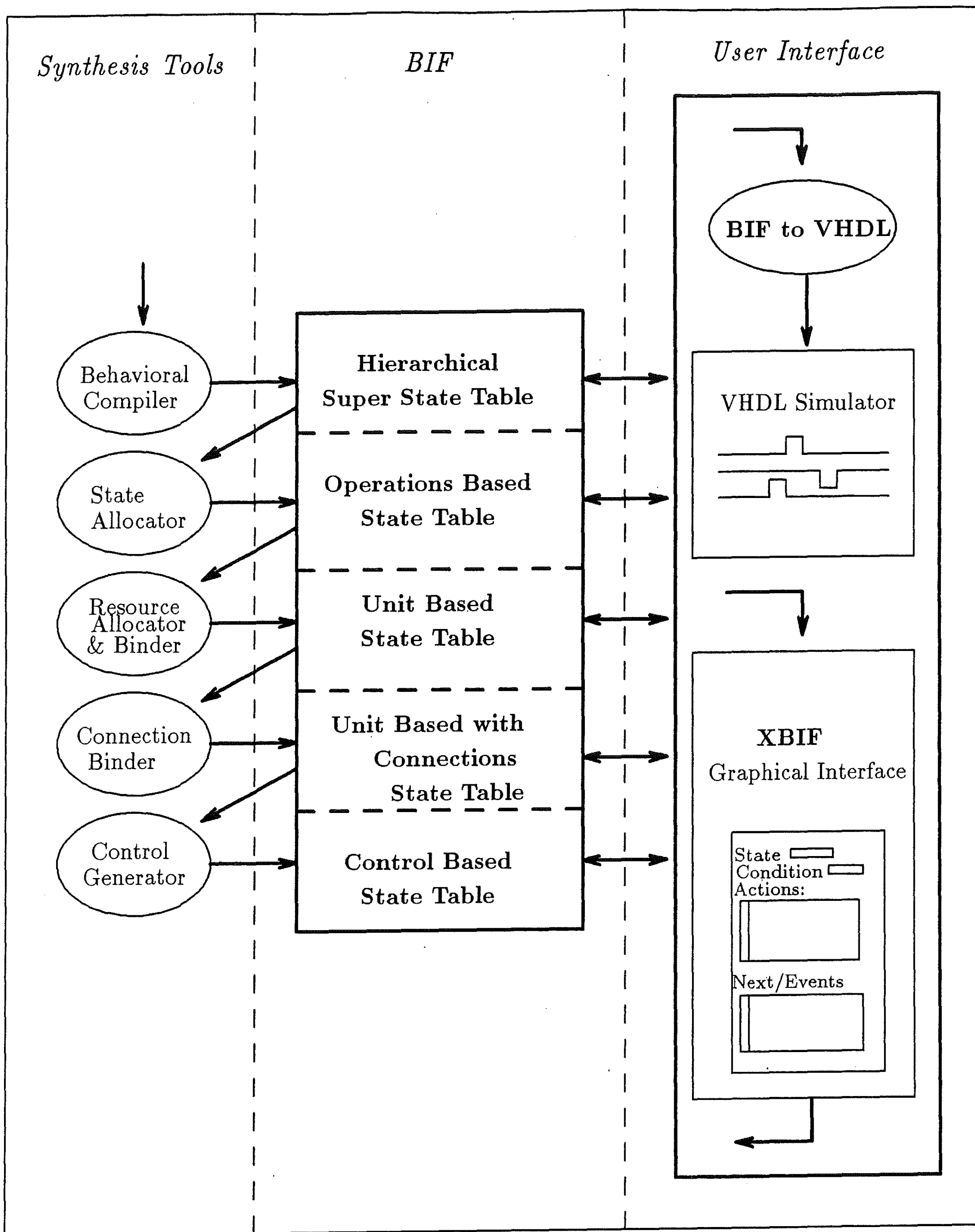


Figure 1: BIF Design Environment

capabilities for any text entered. This helps the user eliminate errors early in design capture. Details of the XBIF user interface can be found in [Hadl90].

The translator from BIF to VHDL is shown in the oval in the upper half of the user interface in Figure 1. The output from the translator can be fed directly to a VHDL simulator such as [Vant89] and observed by the user. The remainder of this report discusses the translation algorithm used by BIF-to-VHDL.

3 Issues in Translating BIF Into VHDL

VHDL does not naturally support the features of behavioral hierarchy, hierarchical events and time-outs, while BIF is well suited to such description styles. Using BIF as a front-end to designing with VHDL thus requires the translation of a BIF description into VHDL. This translation is achieved through the creation and use of special templates and auxiliary signals which facilitate the description of these features in VHDL automatically. This section describes the techniques used to translate some of these constructs from BIF into VHDL, including the basic data structures used, the translation templates, and the role of auxiliary signals used in the VHDL description. The example shown in Figure 2 shows a BIF operations-based description which exhibits a combination of several features including hierarchy, concurrency, event-based transitions and time-outs. This BIF description is used as a running example for the rest of this section.

Although Figure 2 employs only one of the several levels of design abstraction supported by BIF – the operations-based table, the details of the translation algorithm remain largely unchanged for the remaining table formats. The unit-based formats additionally require that VHDL component descriptions exist for each component referenced in a table. The control-based format translation is identical to that of the operations-based format, since the former uses only control inputs to describe actions.

3.1 Representation of Behavioral Hierarchy and Concurrency

The BIF description in Figure 2 shows a behavioral hierarchy where Table *Top_Table* is composed of two sub-tables *h_table* and *c_table*, along with events that effect transitions between any states of these sub-tables. Hierarchy, shown by the arrows between tables, is described naturally by allowing the *Next State* field to contain the name of a sub-table. The *Event* field contains "*(subtable)*" to distinguish the next state from a next state triggered by an event.

Concurrency is indicated by the keyword **Concurrent**, as shown in h_table, which has *a_table* and *b_table* running concurrently under it.

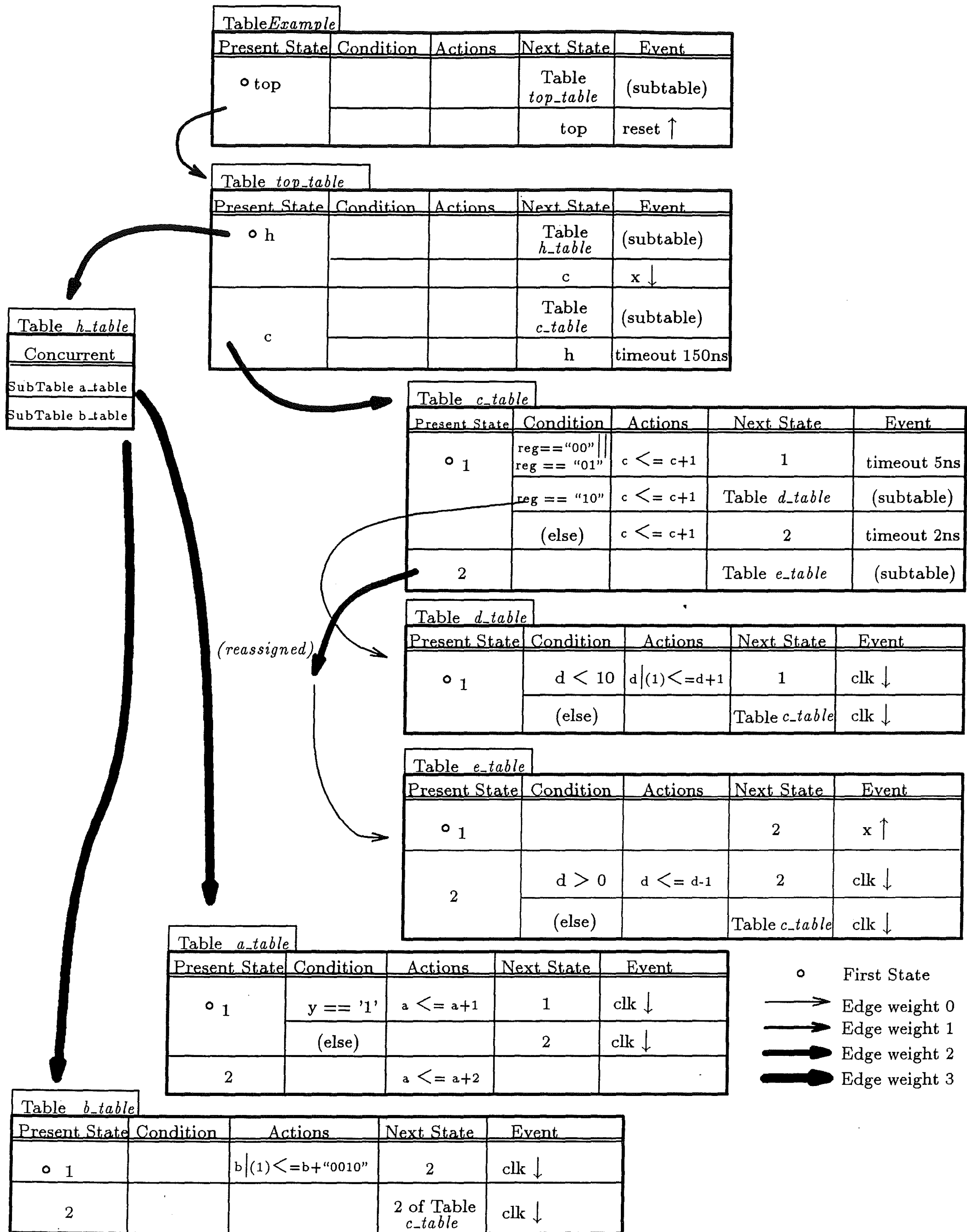


Figure 2: BIF Example

Hierarchy is described in detail in section 3.3.

3.2 Hierarchical Events and Time-outs

Figure 2 shows a hierarchical event "*reset* ↑" at the highest level of description (Table *Example*). If this hierarchical event occurs, it overrides any local event at a lower level. For instance, if the design is in state 1 of e_table, and the events "X ↑" and "*reset* ↑" occur simultaneously, the design would be forced into state *top* of table *Example*.

Time-out events are indicated by the keyword *timeout* and a duration. Time-outs are of two types: global and local. Table top_table shows a global time-out of 150 ns if its present state is c. This implies that a total of 150 ns will be spent in c_table, including any of its states or states of sub-tables, after which control will be transfered to state h of top_table. A local time-out forces a transition to the next state of the present state after a certain duration. For instance, if control is in the first condition|action|next-state triplet of state 1 in c_table, the event "*timeout 5ns*" sequences the design back to state 1 after a duration of 5 ns.

3.3 The Table-Tree

The first step in converting the hierarchical BIF description into VHDL is to capture the hierarchy in a data structure. The *Table-Tree* is one such structure used to represent the hierarchical and concurrent structure of BIF descriptions. It is organized as a multi-way tree with nodes corresponding to BIF tables and edges corresponding to the relationships of hierarchy between tables. The root node is the highest (outermost) level of hierarchy and the leaf nodes represent the lowest (innermost) level of hierarchy. The immediate children of a table node correspond to the sub-tables that the latter defines.

Edges between nodes in the Table-Tree are assigned weights based on the type of hierarchy and/or concurrency that the edge implies. Figure 3 lists all the edge weights and their meanings.

The basic translation algorithm traverses the Table-Tree and generates VHDL code using tem-

Edge Weight	Meaning
0	Conditional hierarchical call
1	Unconditional hierarchical call with 1 substate
2	Unconditional hierarchical call with more than 1 substate
3	Concurrency

Figure 3: Edge Weights

plates that are based on the values of edge weights and the types of sub-trees as described in the following subsections. Figure 4 shows the Table-Tree for the BIF description in Figure 2.

Figure 5 shows a graphical depiction of the hierarchy between the top most table *Example* and table *top_table*. The two tables are represented by correspondingly labeled nodes in Figure 4 separated by an arc weighted with the value 1. This is the simplest form of hierarchy and occurs whenever a parent table has a single state and the condition in that state is assumed to be true. In this example, a next-state event pair is given in table *Example* to indicate a transition on the rising edge of *reset* and to denote the hierarchical relationship between it and *top_table*. When the event occurs, which applies to all tables beneath the parent table, control is returned to the first state of table *Example*, and thus, to the first state of top_table.

The table *top_table* defines two sub-tables, *c_table* and *h_table*, as shown in Figure 6. The edges between the three nodes in the Table-Tree are assigned value 2. This kind of hierarchy differs from type 1 only in that it allows more than one state to define a sub-table.

Figure 7 shows how table *c_table* defines two sub-tables, *d_table* and *e_table*. In state 1 of c_table, d_table is to be a sub-table only if *reg* is equal to the bit value "10". This case identifies conditional hierarchy which is assigned value 0 in the Table-Tree. Notice in Figure 2 that the transition arc from the second state of c_table to e_table actually meets the specification for edge weight 2. However, since c_table has a zero weighted arc from its first state to d_table, the second arc is reassigned

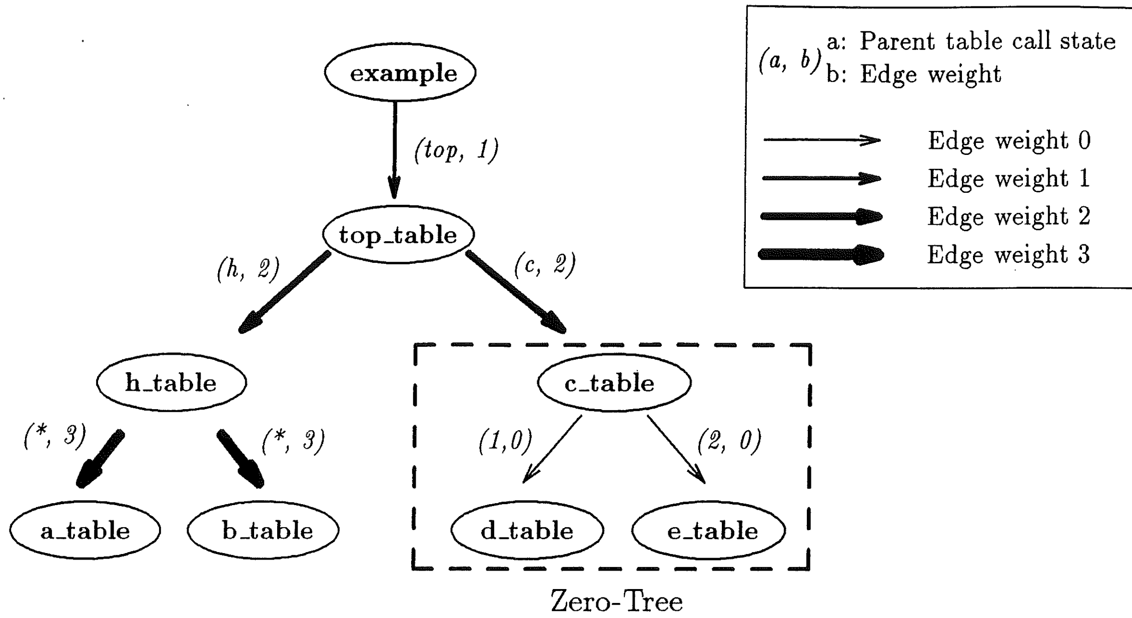


Figure 4: BIF Table-Tree

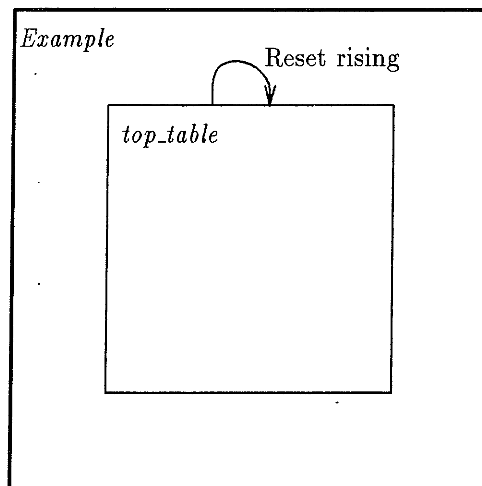


Figure 5: Edge Weight 1

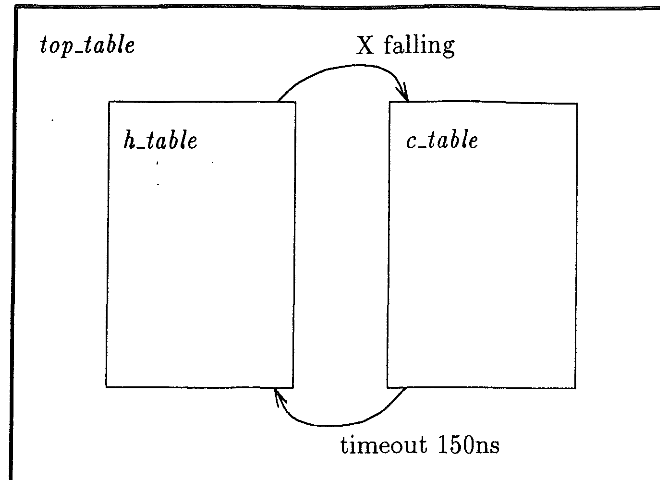


Figure 6: Edge Weight 2

value 0. Together, the three tables become a single Zero-Tree.

3.4 Zero-Trees

After the edge weights have been assigned to the Table-Tree, subtrees identified as *Zero-Trees* are generated and given unique names.

The conditional hierarchy represented by Zero-Trees presents a slight problem for the translation algorithm. If the definition of a sub-table occurs in one condition|action|next-state it should not be "active" under a different condition in the same state. Since the resulting VHDL code uses signals based on state and table names, directly implementing the tree as a hierarchy would cause the sub-table to be activated regardless of the condition. For instance, in Figure 4, c_table is the root of a Zero-Tree with two children, d_table and e_table. The conditional definition of the sub-table d_table requires that it be activated only when c_table is in its first state and the condition *reg == "01"* is true. However, a straightforward translation, mapping tables to VHDL processes, would require that d_table test the same condition. Since the condition occurs in c_table it should only be tested in the VHDL process corresponding to c_table. The solution is to reparent the Zero-Tree with a dummy node and make the original parent node one of the children, as shown in Figure 8.

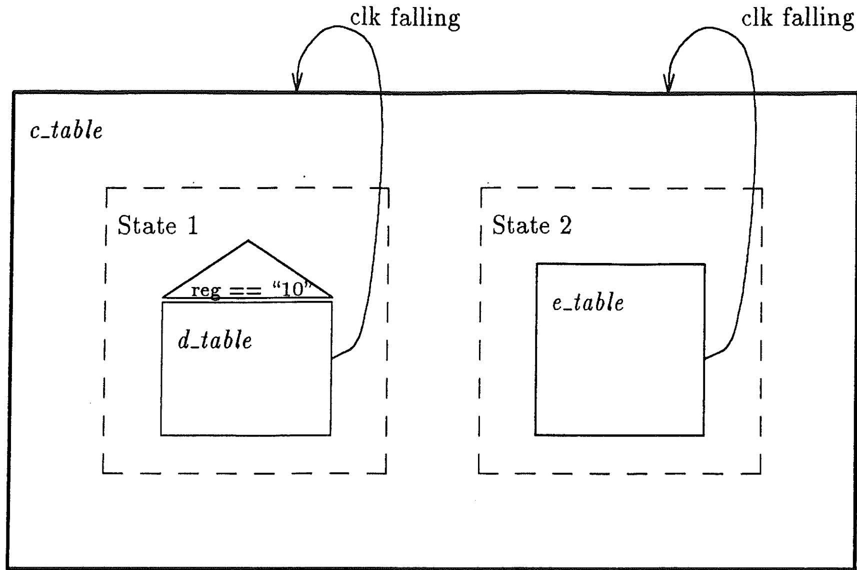


Figure 7: Edge Weight 0

The new node then guarantees correct sequencing behavior between the children.

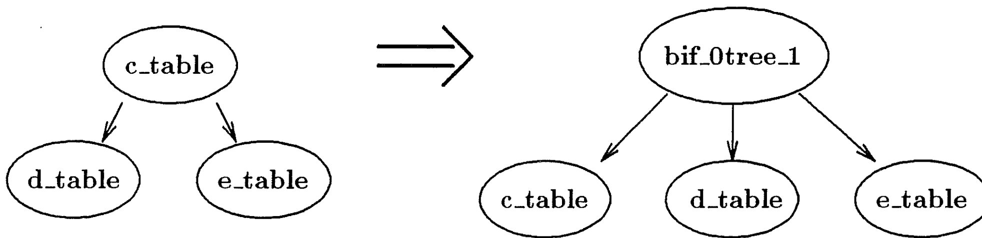


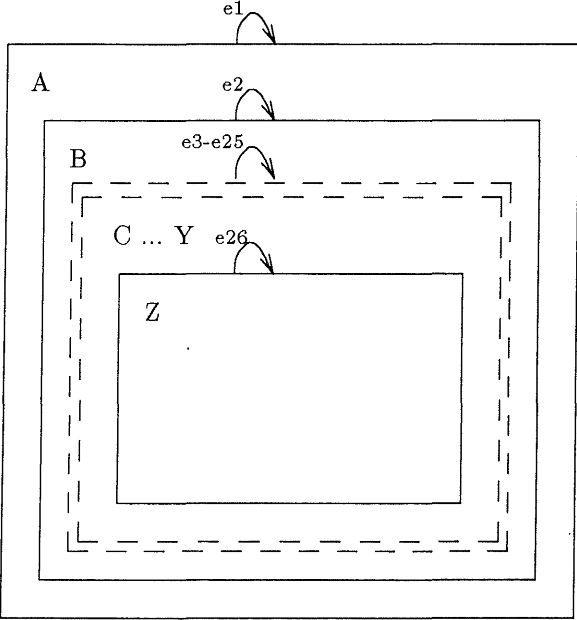
Figure 8: Reparenting a Zero-Tree

3.5 Resolving Hierarchical Events in VHDL

3.5.1 Wait Statements

Although VHDL is capable of representing behavioral hierarchy in a straightforward fashion using wait statements, the resulting description can become very large, repetitive and cumbersome. Since a hierarchical event forces the behavior to enter a default state for all (and any) sub-states, a normal state signal update can be performed only if no hierarchical events "above" the current state have

occurred. The straightforward approach to describing such hierarchy uses wait statements that test for all overriding hierarchical events *at every event point* in the VHDL behavior. This method can easily lead to an explosion in the number of hierarchical events tested and default states assigned. The resulting VHDL code is very long, unreadable and hard to maintain. Figure 9 shows an example of this case. The left-hand side of the figure describes a hierarchy of 26 tables, each having a single event which hierarchically affects all tables beneath it. On the right is the straightforward method for describing this behavior in VHDL. Each table is represented by a VHDL process which waits on all events that might occur in tables at higher levels of hierarchy. The last process, "Z", not only must wait on 26 separate events, but must properly determine which of the 26 events occurred and take appropriate action.




```
A : PROCESS
  BEGIN
    ...
    WAIT UNTIL (e1);
    ...
  END
B: PROCESS
  BEGIN
    ...
   WAIT UNTIL (e1 OR e2);
   IF e1 THEN ...
   ELSEIF e2 THEN ...
   ENDIF
    ...
  END

C: PROCESS
  ...
Y: PROCESS
  ...

Z : PROCESS
 BEGIN
   ...
   WAIT UNTIL (e1 OR e2 OR e3 OR e4, ..., OR e26);
   IF e1 THEN ...
   ELSEIF e2 THEN ...
   ...
   ELSEIF e25 THEN ...
   ELSEIF e26 THEN ...
   ENDIF
   ...
  END
```

HIERARCHICAL DESCRIPTION **VHDL CODE**

Figure 9: Hierarchy Using Wait Statements

Another shortcoming of the vhdl wait statement is the lack of the ability to combine the "wait **on** *event*" statement with the "wait **until** *time-out*". Figure 10 shows a behavioral description of a table that sequences two sub-tables based on a time-out specification. The first sub-table, *A*,

contains two states, *A1* and *A2*, and a transition from A1 to A2 on the occurrence of *EVENT*. The specification for state A1 requires that the design wait in that state for a maximum of 10ns or until the occurrence of EVENT. In VHDL, the desired wait statement for process A1 does not have these semantics and thus does not fulfill the required behavior.

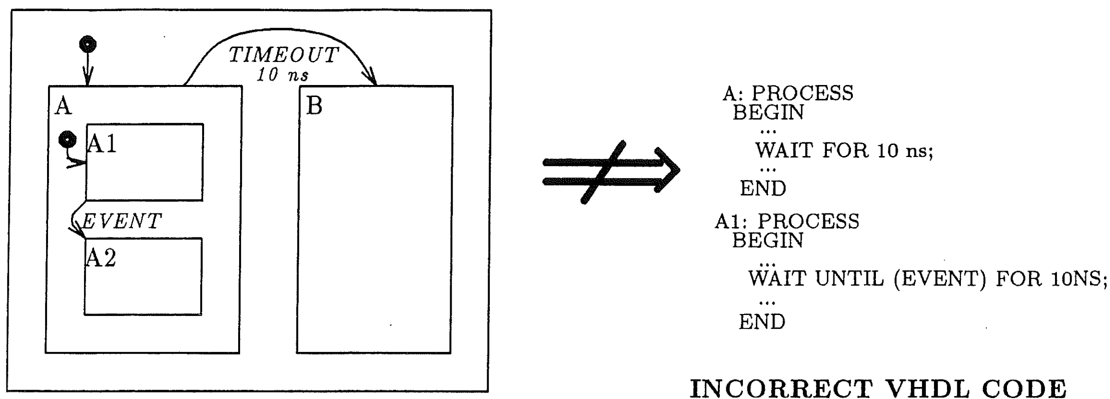


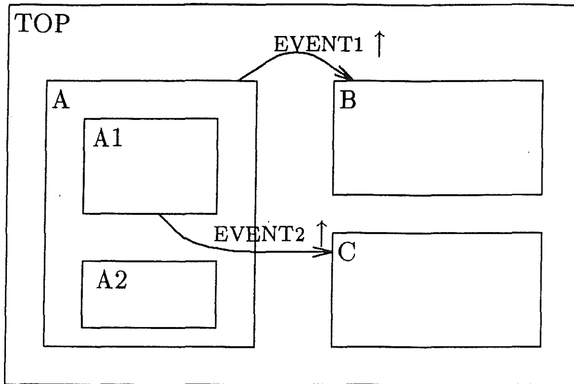
Figure 10: Events and Time-outs In Wait Statements

3.5.2 State Signals and Resolution Functions

The approach taken to avoid the problems using wait statements is as follows. A signal is used to represent the state of the design at each level of the hierarchy. These state signals (along with the signals corresponding to actions performed in that state) can be assigned in many different blocks of the generated VHDL code. Since each of these assignments to state signals represents a state transition at that level of hierarchy, the assignment that corresponds to the highest level of hierarchy should override all other assignments. Hence a VHDL resolution function is written that automatically selects the assignment to the state signal at the highest level of hierarchy.

This is achieved by defining the state signal driver in the resolution function as an array, with assignments to the driver placed in the array in a hierarchical order. By always picking the first element of the array, the state signal is guaranteed to be assigned the state transition corresponding to the highest level of hierarchy.

Figure 11 shows how the described process would work for two tables writing different values to the same signal at the same time.



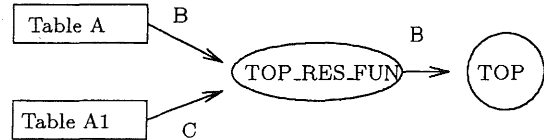

```
A: BLOCK (TOP = A)
  BEGIN
    IF EVENT1='1' and not EVENT1'stable THEN
        TOP <= GUARDED B;
    ENDIF;
  END;

A1: BLOCK(TOP = A AND TABLE_A = A1)
  BEGIN
    IF EVENT2='1' and not EVENT2'stable THEN
        TOP <= GUARDED C;
    ENDIF;
  END;
```

VHDL Code

Table Description

```
TYPE TOP_TABLE IS (A,B,C);
TYPE TOP_RES IS ARRAY(NATURAL RANGE <>) OF TOP_TABLE;
FUNCTION TOP_RES_FUN (INPUT: TOP_RES) RETURN TOP_TABLE IS
  BEGIN
      RETURN INPUT(0);
  END TOP_RES_FUN;
SIGNAL TOP: TOP_RES_FUN TOP_TABLE REGISTER := A;
```

Resolution Function For "TOP"



Signal Resolution For "TOP"

Figure 11: Signal Resolution

Table *TOP* contains two sub-tables *A* and *B*. If the event *EVENT1* ↑ occurs while in A, control is transferred to B. However, if the event *EVENT2* ↑ occurs while in state *A1* of A, control is transferred to C. Since all event transitions that apply to A apply to each of its states hierarchically, when both events occur simultaneously, the first event should "override" the second. The VHDL code on the right shows an abbreviated version of this design. The control context for table TOP is represented by the value of the VHDL signal variable "TOP". Transfer of control to table B on the first event is shown by the statement $TOP <= GUARDEDB$ in the first block. In the second block the transition to table C on the second event is defined in the same manner by $TOP <= GUARDEDC$. When both events occur simultaneously the signal "TOP" can have two possible assignments. Therefore, its resulting value must be resolved. The VHDL resolution function for the signal is shown beneath the table description. When the function is called it unconditionally returns the first element of the list of conflicting values being assigned to "TOP".

Since block A is defined before block A1 in the VHDL code the value the former assigns to "TOP" will precede the value assigned by A1 in the input list. Thus, "TOP" will be assigned the value "B", corresponding to a transfer of control in the design to table B as required by the behavioral description.

This method assumes that processes are evaluated in the VHDL simulator in the order that they are defined. In addition, it requires that the multiple values for a signal be placed in the resolution function's input array in the order that they are received.

For a complete description of VHDL resolution functions see [Arms89].

3.6 VHDL Code Templates

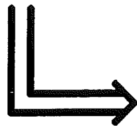
The basic translation strategy generates one VHDL block for every BIF state. The only exceptions are states with global time-outs, which may be assigned a second block to generate the time-out event. (cf Figures 12 and 13). The event is tested for in the first block. Each block may consist of several processes. Two templates, the *Block Template* and the *Block-Process Template*, are used to map BIF states into VHDL code.

The Block Template consists of a VHDL block with a guard at its entry point which tests for the state signal and the event causing the entry to that state. Within the block, guarded signal assignments are used to sequence the state signal to the next state of the design. The Block Template is used for all non-leaf nodes in the Table-Tree, except for those nodes which belong to a Zero-Tree.

Figure 12 shows a table, *P*, containing a state *1* and a global event *RESET*. On the lower right is the VHDL code resulting from the Block Template for this state. The variable *top_table* corresponds to the single sub-table of *P* (the sub-table itself is not shown) and is initialized to its first state by being assigned the enumerated value *top_table_1*. This occurs only when the guard condition is fulfilled, in this case, by the event RESET rising. Since state 1 loops back to itself on the RESET event the variable *bif_self_P_1* is needed to indicate a self transition. The self-variable

and other auxiliary VHDL signals are described in the next section.

Table *P*				
Present State	Condition	Actions	Next State	Event
∘ 1			Table *top_table*	(subtable)
			1	reset ↑

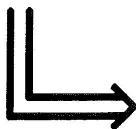


```
P_1: BLOCK (reset='1' and not reset'stable(0 ns))
  BEGIN
    top_table <= guarded top_table_1;
    ...
    bif_self_P_1 <= guarded bif_self_P_1 + 1;
  END BLOCK P_1;
```

Figure 12: An Example Using the Block Template

Figure 13 shows the same table with the transition event changed to a time-out value. In this case, the resulting VHDL code is divided into two separate blocks, the first generating the time-out event and the second testing the event as a guard condition.

Table *P*				
Present State	Condition	Actions	Next State	Event
∘ 1			Table *top_table*	(subtable)
			1	timeout 150ns



```
P_1: BLOCK
BEGIN
   bif_timeout_P_1 <= '0','1' after 150 ns;
END BLOCK P_1;
P_2: BLOCK (bif_timeout_P_1='1' and not bif_timeout_P_1'stable(0 ns))
BEGIN
  top_table <= guarded top_table_1;
  ...
   bif_self_P_2 <= guarded bif_self_P_2 + 1;
END BLOCK P_2;
```

Figure 13: The Block Template with Time-outs

The Block-Process Template is also a VHDL block with a guard at its entry point, but tests

only for the state signal at entry. Two processes are encapsulated within this block: one describing the actions performed in that state, and the other generating the next state information for the current BIF state. The Process Template is used for leaf nodes and Zero-Trees.

Figure 14 shows a table, Q, containing a single state *1* which performs the function $a <= a + 1$ if the condition $reg ==$ "00" is true. Upon completion, control returns to state 1 after 5 ns. On the lower right is the VHDL code resulting from the Block-Process Template for this state. The guard condition is fulfilled only when the variable *table_Q* assumes the enumerated value *table_Q_1*, meaning that the design is in state 1 of table Q. The first process, *Q_1_0*, checks the condition $reg ==$ "00" and performs the required action if the condition is true. The delayed assignment of *bif_timeout_Q_1* simulates an event on that signal after 5 ns. The second process, *Q_1_1*, checks to see if the event on the time-out variable has occured and if so, generates the next state information for this state, in this case, a loop back to itself. As in the previous figure, the variable *bif_self_Q_1* is used to indicate a self transition.

3.7 Auxiliary VHDL Signals

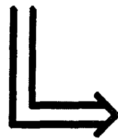
The translation process introduces three types of auxiliary signals.

The global signal *bif_timeout* is used to force time-outs in the generated VHDL description. Figures 13 and 14 show how the signal is generated and tested in each VHDL code template.

States which have transitions back to themselves need to be handled in a special manner, since VHDL's simulation model will not register re-entry into that state unless forced to acknowledge this fact. The global signal *bif_self* is incremented to notify the simulator that a transition back to the same state has occurred. The examples in Figures 12 and 14 both employ the bif_self variable.

BIF's event controlled state sequencing model requires that the value of the condition field of a triplet be tested at the time of entry into a particular state. However, the exiting event (which sequences the design to the next state) may occur at any time after entry into the state. During that time period, the value of the condition tested at entry may change. The translation algorithm

Table Q				
Present State	Condition	Actions	Next State	Event
∘ 1	reg=="00"	a<=a+1	1	timeout 5ns



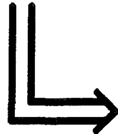
```
Q_1: BLOCK(table_Q = table_Q_1)
BEGIN
  Q_1_0: PROCESS
  BEGIN
    IF GUARD THEN
       if reg=b"00" then
         a <= a+1 ;
         bif_timeout_Q_1 <= '0','1' after 5 ns;
       end if;
    END IF;
  WAIT ON GUARD, bif_self_Q_1;
  END PROCESS Q_1_0;
  Q_1_1: PROCESS
  BEGIN
    IF GUARD THEN
     if bif_timeout_Q_1='1' and not bif_timeout_Q_1'stable then
         bif_self_Q_1 <= bif_self_Q_1+1;
     end if;
    END IF;
  WAIT ON GUARD, bif_timeout_Q_1;
  END PROCESS Q_1_1;
END BLOCK Q_1;
```

Figure 14: An Example Using the Block-Process Template

uses a local signal, *bif_case*, to store the value of the condition that was tested at the time of entry into the state. Figure 15 shows an example similar to the one in Figure 14 but which contains an additional condition|action|next-state triplet in state 1. If the condition *reg* == "11" is true the action decrements the value of *a* and the design waits on the event "*reset* ↑" before looping back to itself. Since there are now multiple conditions in this state, the first process, *Q_1_0*, assigns the bif_case variable a value to indicate which condition was true and the second process, *Q_1_1*, checks this value and waits for a change on either signal *bif_timeout_Q_1* or *reset*.

Table Q

Present State	Condition	Actions	Next State	Event
○ 1	reg=="00"	a<=a+1	1	timeout 5ns
	reg=="11"	a<=a-1	1	reset ↑



```
Q_1: BLOCK(table_Q = table_Q_1)
SIGNAL bif_case: integer:=0;
BEGIN
  Q_1_0: PROCESS
  BEGIN
    IF GUARD THEN
      if reg=b"00" then
        a <= a+1 ;
        bif_timeout_Q_1 <= '0','1' after 5 ns;
        bif_case <= 1;
      else if reg=b"11" then
        a <= a-1 ;
        bif_case <= 2;
      end if;
    END IF;
  WAIT ON GUARD, bif_self_Q_1;
  END PROCESS Q_1_0;
  Q_1_1: PROCESS
  BEGIN
    IF GUARD THEN
      if bif_case=1 then
        if bif_timeout_Q_1='1' and not bif_timeout_Q_1'stable then
            bif_self_Q_1 <= bif_self_Q_1+1;
        end if;
      else if bif_case=2 then
        if reset='1' and not reset'stable then
            bif_self_Q_1 <= bif_self_Q_1+1;
        end if;
      end if;
    END IF;
  WAIT ON GUARD, bif_timeout_Q_1, reset;
  END PROCESS Q_1_1;
END BLOCK Q_1;
```

Figure 15: The Use of the *bif_case* Variable

3.8 Overall Structure of Translated VHDL Code

Figure 16 shows the overall structure of the VHDL code translated from BIF. The generated VHDL description begins with the *library* specification for any packages that may be used in the VHDL description. This is followed by the *entity* part which describes the primary inputs declared in BIF's symbol list. The *architecture* part contains a number of behavioral VHDL blocks, with each block roughly corresponding to a state in the BIF representation. Each such block is defined by either the Block Template or the Block-Process Template as described earlier. The generated VHDL code ends with a small *configuration* part.

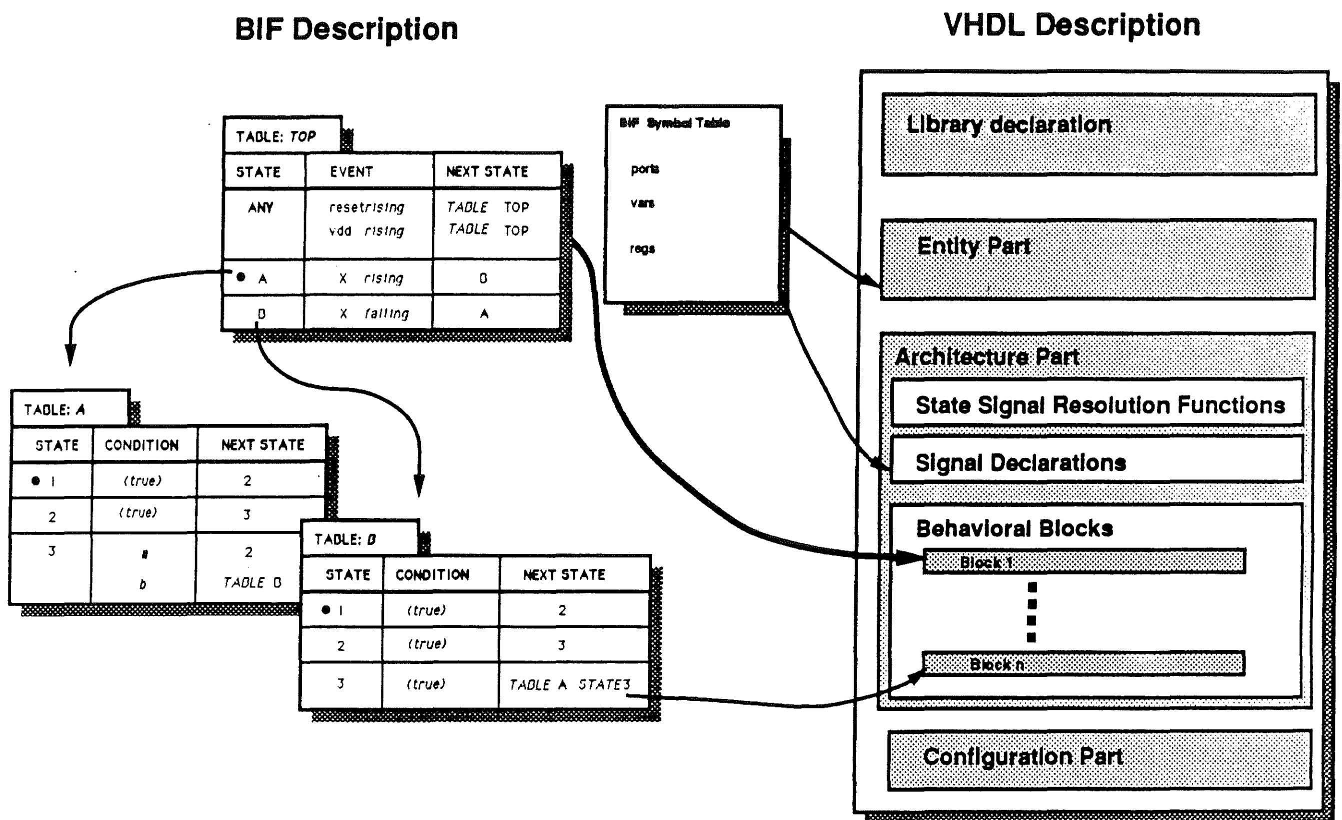


Figure 16: Organization of Translated VHDL Code.

4 The Translation Algorithm

4.1 Outline of Translation Algorithm

The basic strategy used to perform the translation is to capture the hierarchical structure of the behavior in a Table-Tree and apply translation templates to this hierarchical structure in order to generate the behavioral VHDL code.

Appendix I contains the VHDL code that was automatically generated using this algorithm for the BIF description shown in Figure 5.

Step 1. Create Module Names in VHDL. The BIF file name is appended with the characters "_e", "_a" and "_c" for the entity, architecture and configuration parts of the translated VHDL code.

Step 2. Generate Signal Names. BIF's symbol list is scanned to convert BIF variables and ports into equivalent VHDL carriers. Primary inputs in BIF are translated into primary inputs in VHDL. Primary outputs and internal variables in BIF are translated into VHDL internal signals. If an internal signal is assigned in more than one state, the signal is marked as a resolved signal.

Step 3. Generate Hierarchical Table Tree. The BIF hierarchical description is scanned to generate its equivalent Table Tree. Edge weights are assigned to the tree based on the type of hierarchical call.

Step 4. Generate and Extract Zero-Trees. In a second pass of the Table Tree, Zero-Trees are created by assigning all siblings of a zero-edge a weight of zero. Each Zero-Tree is assigned a unique name, and a list of all Zero-Trees is created.

Step 5. List All Tables and States. Each BIF table and Zero-Tree is scanned to generate a list of names in that table or tree. Default entry states are marked.

Step 6. Generate Architecture Table. An internal tabular representation of the BIF description is created using an *Architecture Table.* This table is similar to a BIF description, except that the current state and next state fields contain (*state-tuples*) pairs which specify the table name and the state name for the BIF description.

Step 7. Split Global Time-Out States. Each State in the Architecture Table that has a Global Time-Outs is split into two states: the first which eventually creates the VHDL time-out signal, and the second which checks for the Time-Out condition.

Step 8. Assign VHDL Template. VHDL templates are assigned to states of the Architecture Table based on their position in the Table-Tree. Non-leaf nodes are mapped into Template1 blocks, while leaf nodes are mapped into Template2 blocks.

Step 9. Fully Qualify State-Tuples. The state-tuples in the Architecture Table represent state information local to that level of hierarchy only. In this step, each state-tuple in the Architecture Table is fully qualified by specifying global state information.

For current state-tuple entries in the Architecture Table, a bottom-up search is performed to generate state-signal names for every level to the root of the Table-Tree. For next state-tuple entries in the Architecture Table, a top-down search from the root to the state entry is performed to generate state-tuples for intermediate levels.

Step 10. Generate Auxiliary Signals. For any state which has a self-loop (where next-tuple = present-tuple), a auxiliary signal of the form *bif_self_<block_label>* is created. Similarly, a auxiliary signal of the form *bif_timeout_<block_label>* is created for any state that has a global or local time-out. If any state has triplets with multiple conditions, a auxiliary local signal of the form *bif_case* is created for the corresponding VHDL block.

Step 11. Generate VHDL code. If the BIF description uses constructs such as bit-field manipulation, the corresponding VHDL package has to be loaded. Hence a VHDL library declaration is first generated where necessary. Next, the entity description for the VHDL code is generated by declaring all primary inputs from Step 2. The VHDL Architectural

body is generated next. The body begins with the definition of state-signals derived from the state-list in Step 5. This is followed by a definition of internal signal names from the symbol table in Step 2 and the global auxiliary signals (bif_self and bif_timeout) in Step 10. The remainder of the Architectural body consists of the VHDL templates selected in Step 8. Within each generated VHDL template, the local signal bif_case is declared if used in that block. Finally, a dummy configuration body is created.

5 A Detailed Example

The following example traces each step of the BIF to VHDL algorithm for the example shown in Figure 2.

Step 1. Derive names.

Entity name: example_e
Architecture name: example_a
Configuration name: example_c

Step 2. Define a symbol table.

Symbol Name	*Port Type*	*Symbol Type*	*MSB*	*LSB*	*Resolved*
reset	in	bit			
x	in	bit			
y	in	bit			
reg	in	bit_vector	1	0	
clk	in	bit			
a	signal	integer			yes
b	signal	bit_vector	3	0	no
c	signal	integer			no
d	signal	integer			yes

Step 3. Form hierarchy table tree.

	example	*top_table*	*h_table*	*c_table*	*a_table*	*b_table*	*d_table*	*e_table*
example	no_edge	(top,1)	no_edge	no_edge	no_edge	no_edge	no_edge	no_edge
top_table	no_edge	no_edge	(h,2)	(c,2)	no_edge	no_edge	no_edge	no_edge
h_table	no_edge	no_edge	no_edge	no_edge	(*,3)	(*,3)	no_edge	no_edge
c_table	no_edge	no_edge	no_edge	no_edge	no_edge	no_edge	(1,0)	(1,0)
a_table	no_edge	no_edge	no_edge	no_edge	no_edge	no_edge	no_edge	no_edge
b_table	no_edge	no_edge	no_edge	no_edge	no_edge	no_edge	no_edge	no_edge
d_table	no_edge	no_edge	no_edge	no_edge	no_edge	no_edge	no_edge	no_edge
e_table	no_edge	no_edge	no_edge	no_edge	no_edge	no_edge	no_edge	no_edge

Step 4. Derive zero-trees.

Tree Name	*Tree Type*	*Element Table*
bif_0tree_1	0	c_table, d_table, e_table

Step 5. Generate state lists.

Table Name	*Element State Name*
top_table	h, (c, bif_0tree_1)
bif_0tree_1	c_table, d_table, e_table
a_table	1,2
b_table	1,2
c_table	1,2
example	top
d_table	1
e_table	1

Step 6. Generate an architecture table.

Present State	*Condition*	*Actions*	*Next State*	*Event*
(example,top)	true	null	(example,top)	(reset,rising)
(top_table,h)	true	null	(top_table,c)	(x,falling)
(top_table,c)	true	null	(top_table,h)	(150 ns)
(a_table,1)	y=='1'	$a\vert(1) <= a+1$	(a_table,1)	(clk,falling)
	else	null	(a_table,2)	(clk,falling)
(a_table,2)	true	$a\vert(1) <= a+2$	(a_table,1)	(clk,falling)
(b_table,1)	true	$b\vert(1) <= b+\text{"0010"}$	(b_table,2)	(clk,falling)
(b_table,2)	true	null	(c_table,2)	(clk,falling)
(c_table,1)	$reg == \text{"00"} \Vert reg == \text{"01"}$	$c\vert(1) <= c+1$	(c_table,1)	(5 ns)
	$reg == \text{"10"}$	$c\vert(1) <= c+1$	(d_table,*)	(subtable)
	else	$c\vert(1) <= c+1$	(c_table,2)	(2 ns)
(c_table,2)	true	null	(e_table,*)	(x,rising)
(d_table,1)	$d < 10$	$d\vert(1) <= d+1$	(d_table,1)	(clk,falling)
	else	null	(c_table,*)	(clk,falling)
(e_table,1)	$d > 0$	$d\vert(1) <= d-1$	(e_table,1)	(clk,falling)
	else	null	(c_table,*)	(clk,falling)

Step 7. Assign a template type to each state (*B* for Block Template, *BP* for Block-Process Template).

Step 8. Replace Present-State and Next State fields with tuple representing table and state name

Step 9. Generate auxiliary signals.

Present State	*Condition*	*Actions*	*Next State*	*Event*	*Template*
null	true	null	(top_table,h) (a_table,1) (b_table,1) self	(reset,rising)	B
(top_table,h)	true	null	(top_table,bif_0tree_1) (bif_0tree_1,c_table) (c_table,1)	(x,falling)	B
(top_table,bif_0tree_1)	true	(timeout,150ns)			BP
(top_table,bif_0tree_1)	true	null	(top_table,h) (a_table,1) (b_table,1)	(timeout,rising)	B
(top_table,h) (a_table,1)	$y ==' 1'$	$a\vert(1) <= a + 1$	self	(clk,falling)	BP
	else	null	(a_table,2)	(clk,falling)	
(top_table,h) (a_table,2)	true	$a\vert(1) <= a + 2$	(a_table,1)	(clk,falling)	BP
(top_table,h) (b_table,1)	true	$b\vert(1) <= b + "0010"$	(b_table,2)	(clk,falling)	BP
(top_table,h)	true	null	(top_table,bif_0tree_1) (bif_0tree_1,c_table) (c_table,2)	(clk,falling)	BP
(top_table,bif_0tree_1) (bif_0tree_1,c_table) cline2-5 (c_table,1)	$reg == "00"$ $\Vert reg == "10"$ $reg == "10"$	(timeout,5 ns) null	self (bif_0tree_1,d_table)	(timeout,rising) (subtable)	BP
	else	(timeout, 2 ns)	(c_table,2)	(timeout, rising)	
(top_table,bif_0tree_1) (bif_0tree_1,c_table) (c_table,2)	true	null	(bif_0tree_1,e_table)	(x,rising)	BP
(top_table,bif_0tree_1) (bif_0tree,d_table)	$d < 10$	$d\vert(1) <= d + 1$	self	(clk,falling)	BP
	else	null	(bif_0tree_1,c_table) (c_table,1)	(clk,falling)	
(top_table,bif_0tree_1) (bif_0tree,e_table)	$d > 0$	$d\vert(1) <= d - 1$	self	(clk,falling)	BP
	else	null	(bif_0tree_1,c_table) (c_table,1)	(clk,falling)	

Step 10. Generate an entity statement.

```
LIBRARY PACK;
USE PACK.DEFS.ALL;

ENTITY example_e IS
   PORT (reset: in bit;
         x : in bit;
         y : in bit;
         clk: in bit;
         reg: in bit_vector(1 downto 0));
END example_e;
```

Step 11. Generate an architecture statement.

(1) Architecture name.

```
ARCHITECTURE example_a OF example_e IS
```

(2) Output state signal declarations from Step 5.

```
TYPE TABLE_top_table IS (top_table_h,top_table_bif_0tree_1);
TYPE top_table_RES IS ARRAY(NATURAL RANGE <>) OF TABLE_top_table;
FUNCTION top_table_RES_FUN (INPUT: top_table_RES) RETURN TABLE_top_table IS
BEGIN
   RETURN INPUT(0);
END top_table_RES_FUN;
SIGNAL top_table: top_table_RES_FUN TABLE_top_table REGISTER := top_table_h;

TYPE TABLE_a_table IS (a_table_1,a_table_2);
TYPE a_table_RES IS ARRAY(NATURAL RANGE <>) OF TABLE_a_table;
FUNCTION a_table_RES_FUN (INPUT: a_table_RES) RETURN TABLE_a_table IS
BEGIN
   RETURN INPUT(0);
END a_table_RES_FUN;
SIGNAL a_table: a_table_RES_FUN TABLE_a_table REGISTER := a_table_1;

TYPE TABLE_b_table IS (b_table_1,b_table_2);
TYPE b_table_RES IS ARRAY(NATURAL RANGE <>) OF TABLE_b_table;
FUNCTION b_table_RES_FUN (INPUT: b_table_RES) RETURN TABLE_b_table IS
BEGIN
   RETURN INPUT(0);
END b_table_RES_FUN;
SIGNAL b_table: b_table_RES_FUN TABLE_b_table REGISTER := b_table_1;

TYPE TABLE_bif_0tree_1 IS (bif_0tree_1_c_table,bif_0tree_1_d_table,
   bif_0tree_1_e_table);
TYPE bif_0tree_1_RES IS ARRAY(NATURAL RANGE <>) OF TABLE_bif_0tree_1;
FUNCTION bif_0tree_1_RES_FUN (INPUT: bif_0tree_1_RES) RETURN
   TABLE_bif_0tree_1 IS
BEGIN
   RETURN INPUT(0);
END bif_0tree_1_RES_FUN;
SIGNAL bif_0tree_1: bif_0tree_1_RES_FUN TABLE_bif_0tree_1
   REGISTER := bif_0tree_1_c_table;

TYPE TABLE_c_table IS (c_table_1,c_table_2);
TYPE c_table_RES IS ARRAY(NATURAL RANGE <>) OF TABLE_c_table;
FUNCTION c_table_RES_FUN (INPUT: c_table_RES) RETURN TABLE_c_table IS
BEGIN
   RETURN INPUT(0);
END c_table_RES_FUN;
SIGNAL c_table: c_table_RES_FUN TABLE_c_table REGISTER := c_table_1;
```

(3) Output signals of port type "SIGNAL" from the symbol table.

```
TYPE a_RES IS ARRAY(NATURAL RANGE <>) OF integer;
FUNCTION a_RES_FUN (INPUT: a_RES) RETURN integer IS
BEGIN
   RETURN INPUT(0);
END a_RES_FUN;
SIGNAL a: a_RES_FUN integer REGISTER := 0;

TYPE d_RES IS ARRAY(NATURAL RANGE <>) OF integer;
FUNCTION d_RES_FUN (INPUT: d_RES) RETURN integer IS
BEGIN
   RETURN INPUT(0);
END d_RES_FUN;
SIGNAL d: d_RES_FUN integer REGISTER := 0;

signal b: bit_vector(3 downto 0) := b"0000";
signal c: integer := 0;
```

(4) Output auxiliary variables.

```
signal bif_self_lb_0: integer := 0;
signal bif_self_lb_4: integer := 0;
signal bif_self_lb_8: integer := 0;
signal bif_self_lb_10: integer := 0;
signal bif_self_lb_11: integer := 0;
signal bif_timeout_lb_2: bit := '0';
signal bif_timeout_lb_8: bit := '0';
```

(5) Output the architecture body from the architecture table in Step 9.

```
BEGIN
LB_0: BLOCK (reset='1' and not reset'stable(0 ns))
BEGIN
   top_table <= guarded top_table_h;
   a_table <= guarded a_table_1;
   b_table <= guarded b_table_1;
   bif_0tree_1 <= guarded bif_0tree_1;
   c_table <= guarded c_table;
   bif_self_lb_0 <= guarded bif_self_lb_0 + 1;
END BLOCK LB_0;

LB_1: BLOCK (top_table=top_table_h and (x='0' and not x'stable(0 ns)))
BEGIN
   top_table <= guarded top_table_bif_0tree_1;
   a_table <= guarded a_table;
   b_table <= guarded b_table;
   bif_0tree_1 <= guarded bif_0tree_1_c_table;
   c_table <= guarded c_table_1;
END BLOCK LB_1;

LB_2: BLOCK (top_table=top_table_bif_0tree_1)
BEGIN
   LB_2_0: PROCESS
   BEGIN
      IF GUARD THEN
         bif_timeout_lb_2 <= '0','1' after 150 ns;
      ELSE
      END IF;
   WAIT ON GUARD;
   END PROCESS LB_2_0;
END BLOCK LB_2;

LB_3: BLOCK (top_table=top_table_bif_0tree_1 and (bif_timeout_lb_2='1'
   and not bif_timeout_lb_2'stable(0 ns)))
BEGIN
   top_table <= guarded top_table_h;
   a_table <= guarded a_table_1;
   b_table <= guarded b_table_1;
   bif_0tree_1 <= guarded bif_0tree_1;
   c_table <= guarded c_table;
END BLOCK LB_3;

LB_4: BLOCK (top_table=top_table_h and a_table=a_table_1)
   signal bif_case: integer := 0;
BEGIN
   LB_4_0: PROCESS
   BEGIN
      IF GUARD THEN
         if y='1' then
            a <= transport a + 1 after 1 ns;
            bif_case <= 1;
```

```
         else
            bif_case <= 2;
         end if;
      ELSE
         a <= null;
      END IF;
   WAIT ON GUARD,bif_self_lb_0,bif_self_lb_4;
   END PROCESS LB_4_0;
   LB_4_1: PROCESS
   BEGIN
      IF GUARD THEN
         if not (reset='1' and not reset'stable) then
            if bif_case=1 then
               if clk='0' and not clk'stable then
                  a_table <= a_table_1;
                  bif_self_lb_4 <= bif_self_lb_4 + 1;
               end if;
            elsif bif_case=2 then
               if clk='0' and not clk'stable then
                  a_table <= a_table_2;
               end if;
            end if;
         end if;
      ELSE
         a_table <= null;
      END IF;
   WAIT ON GUARD,clk;
   END PROCESS LB_4_1;
END BLOCK LB_4;

LB_5: BLOCK (top_table=top_table_h and a_table=a_table_2)
BEGIN
   LB_5_0: PROCESS
   BEGIN
      IF GUARD THEN
         a <= transport a + 2 after 1 ns;
      ELSE
         a <= null;
      END IF;
   WAIT ON GUARD;
   END PROCESS LB_5_0;
   LB_5_1: PROCESS
   BEGIN
      IF GUARD THEN
         if clk='0' and not clk'stable then
            a_table <= a_table_1;
         end if;
      ELSE
         a_table <= null;
      END IF;
   WAIT ON GUARD,clk;
   END PROCESS LB_5_1;
END BLOCK LB_5;

LB_6: BLOCK (top_table=top_table_h and b_table=b_table_1)
BEGIN
   LB_6_0: PROCESS
   BEGIN
      IF GUARD THEN
         b <= transport b + b"0010" after 1 ns;
      END IF;
   WAIT ON GUARD,bif_self_lb_0;
   END PROCESS LB_6_0;
   LB_6_1: PROCESS
   BEGIN
      IF GUARD THEN
         if not (reset='1' and not reset'stable) then
            if clk='0' and not clk'stable then
```

```
            b_table <= b_table_2;
         end if;
      end if;
   ELSE
      b_table <= null;
   END IF;
 WAIT ON GUARD,clk;
 END PROCESS LB_6_1;
END BLOCK LB_6;

LB_7: BLOCK (top_table=top_table_h and b_table=b_table_2)
BEGIN
   LB_7_1: PROCESS
   BEGIN
      IF GUARD THEN
         if clk='0' and not clk'stable then
            top_table <= top_table_bif_0tree_1;
            bif_0tree_1 <= bif_0tree_1_c_table;
            c_table <= c_table_2;
         end if;
      ELSE
         top_table <= null;
         bif_0tree_1 <= null;
         c_table <= null;
      END IF;
   WAIT ON GUARD,clk;
   END PROCESS LB_7_1;
END BLOCK LB_7;

LB_8: BLOCK (top_table=top_table_bif_0tree_1 and
   bif_0tree_1=bif_0tree_1_c_table and c_table=c_table_1)
   signal bif_case: integer := 0;
BEGIN
   LB_8_0: PROCESS
   BEGIN
      IF GUARD THEN
         c <= transport c + 1 after 1 ns;
         if reg=b"00" or reg=b"01" then
            bif_timeout_lb_8 <= '0','1' after 2 ns;
            bif_case <= 1;
         elsif reg=b"10" then
            bif_0tree_1 <= bif_0tree_1_d_table;
            bif_case <= 2;
         else
            bif_timeout_lb_8 <= '0','1' after 2 ns;
            bif_case <= 3;
         end if;
      ELSE
         bif_0tree_1 <= null;
      END IF;
   WAIT ON GUARD,bif_self_lb_8;
   END PROCESS LB_8_0;
   LB_8_1: PROCESS
   BEGIN
      IF GUARD THEN
         if bif_case=1 then
            if bif_timeout_lb_8='1' and not bif_timeout_lb_8'stable then
               c_table <= c_table_1;
               bif_self_lb_8 <= bif_self_lb_8 + 1;
            end if;
         elsif bif_case=3 then
            if bif_timeout_lb_8='1' and not bif_timeout_lb_8'stable then
               c_table <= c_table_2;
            end if;
         end if;
      ELSE
         c_table <= null;
      END IF;
```

```
    WAIT ON GUARD,bif_timeout_lb_8;
    END PROCESS LB_8_1;
 END BLOCK LB_8;

 LB_9: BLOCK (top_table=top_table_bif_0tree_1 and
    bif_0tree_1=bif_0tree_1_c_table and c_table=c_table_2)
 BEGIN
    LB_9_1: PROCESS
    BEGIN
       IF GUARD THEN
          if x='1' and not x'stable then
             bif_0tree_1 <= bif_0tree_1_e_table;
          end if;
       ELSE
          bif_0tree_1 <= null;
       END IF;
    WAIT ON GUARD,x;
    END PROCESS LB_9_1;
 END BLOCK LB_9;

 LB_10: BLOCK (top_table=top_table_bif_0tree_1 and
    bif_0tree_1=bif_0tree_1_d_table)
    signal bif_case: integer := 0;
 BEGIN
    LB_10_0: PROCESS
    BEGIN
       IF GUARD THEN
          if d < 10 then
             d <= transport d + 1 after 1 ns;
             bif_case <= 1;
          else
             bif_case <= 2;
          end if;
       ELSE
          d <= null;
          d <= null;
       END IF;
    WAIT ON GUARD,bif_self_lb_10;
    END PROCESS LB_10_0;
    LB_10_1: PROCESS
    BEGIN
       IF GUARD THEN
          if bif_case=1 then
             if clk='0' and not clk'stable then
                bif_self_lb_10 <= bif_self_lb_10 + 1;
             end if;
          elsif bif_case=2 then
             if clk='0' and not clk'stable then
                bif_0tree_1 <= bif_0tree_1_c_table;
                c_table <= c_table_1;
             end if;
          end if;
       ELSE
          bif_0tree_1 <= null;
          c_table <= null;
       END IF;
    WAIT ON GUARD,clk;
    END PROCESS LB_10_1;
 END BLOCK LB_10;

 LB_11: BLOCK (top_table=top_table_bif_0tree_1 and
    bif_0tree_1=bif_0tree_1_e_table)
    signal bif_case: integer := 0;
 BEGIN
    LB_11_0: PROCESS
    BEGIN
       IF GUARD THEN
          if d > 0 then
```

```
            d <= transport d - 1 after 1 ns;
            bif_case <= 1;
         else
            bif_case <= 2;
         end if;
      ELSE
         d <= null;
      END IF;
   WAIT ON GUARD,bif_self_lb_11;
   END PROCESS LB_11_0;
   LB_11_1: PROCESS
   BEGIN
      IF GUARD THEN
         if bif_case=1 then
            if clk='0' and not clk'stable then
               bif_self_lb_11 <= bif_self_lb_11 + 1;
            end if;
         elsif bif_case=2 then
            if clk='0' and not clk'stable then
               bif_Otree_1 <= bif_Otree_1_c_table;
               c_table <= c_table_1;
            end if;
         end if;
      ELSE
         bif_Otree_1 <= null;
         c_table <= null;
      END IF;
   WAIT ON GUARD,clk;
   END PROCESS LB_11_1;
END BLOCK LB_11;

END example_a;
```

Step 12. Output the configuration statement.

```
CONFIGURATION example_c OF example_e IS
   FOR example_a
   END FOR;
END example_c;
```

6 Experiments

The BIF-to-VHDL translation algorithm was used to generate VHDL code for several designs exhibiting a variety of design features, as shown in Figure 17. The size of each BIF description is indicated in a column by the number of words used.

On the examples tested, the size of the BIF descriptions are between 12.4% and 23.5% of the corresponding VHDL descriptions, with an average ratio of 16.7%. The BIF descriptions are therefore much more concise in describing complex behavior.

It should be noted that the translated VHDL code for these examples is actually quite small in size as compared to VHDL code written in a straightforward fashion using wait statements. This decrease in the size of the VHDL code is achieved through the novel method of controlling the explosion in hierarchical events using the resolution function scheme as described in Section 3. Furthermore, the translated VHDL code has high readability since there is almost a one-to-one correspondence to states described in the BIF description.

Example	Description	BIF size (words)	VHDL size (words)	BIF/VHDL (ratio)	Hierarchy	Concurrency	Global Timeout	Local Timeout	Asynch Events	Clocked Transitions
bus_arbiter	PLD Hb	152	808	0.188						•
mano_139	FSM	82	510	0.160						•
tlc	Trfc Lt Cont	81	628	0.129	•	•		•	•	
cc	Cntrl Counter	203	864	0.235	•			•	•	
tec14	Hierarchy	106	521	0.204	•				•	
tec17	Concurrency	81	494	0.164	•	•			•	
tec26	Struct Mod	58	365	0.159						•
tec34	Quotient Acc	70	507	0.138					•	
tec38	Bus Mem Ctl	67	540	0.124					•	
example_1	LS Timeout	88	491	0.179	•			•	•	
example_2	GI Timeout	86	493	0.174	•		•		•	
example_3	Hier & Concur	113	748	0.151	•	•		•	•	
ICCAD_90	Figure 5	192	1120	0.171	•	•	•	•	•	

Figure 17: Experiments

7 Summary

This report described an algorithm for automatically translating BIF into behavioral VHDL code. The issues that were addressed highlighted some of the deficiencies of VHDL when trying to describe behavioral hierarchy, hierarchical events, and time-outs. We presented BIF as a good input path to VHDL and introduced methods for efficient representation for simulation of complex hierarchical behavior in VHDL.

The BIF to VHDL translator was tested on a number of examples and the results showed a greater measure of conciseness in BIF design descriptions over VHDL.

8 Acknowledgements

This work was supported in part by NSF grant MIP-9009239, in part by the state of California MICRO grant 89-057, and in part by a grant from KOSEF and ETRI (Korea).

The authors would also like to thank Prof. Dan Gajski for his comments.

9 References

References

[Arms89] Armstrong, J., "Chip Level Modeling with VHDL," Prentice Hall, 1989.

[DuHG89] Dutt, N., Hadley, T., and Gajski, D., "BIF: A Behavioral Intermediate Format For High Level Synthesis," Tech. Rpt. # 89-03, UC Irvine, September, 1989.

[DuHG90] Dutt, N., Hadley, T., and Gajski, D., "An Intermediate Representation for Behavioral Synthesis," *27th Design Automation Conference*, June, 1990.

[Hadl90] Hadley, T., "The BIF User Interface and Programming Manual," Tech. Rpt. # 90-07, UC Irvine, April 1990.

[Vant89] Vantage Analysis Systems, Inc, Fremont, CA 1989.

[VHDL87] *IEEE Standard VHDL Language Reference Manual*, IEEE, 1987.

A Appendix

A.1 Technical Report Example: BIF

```
SYMBOL_TABLE {

   type
      BOOLEAN   = {0};
      EVENT     = {0};
      TWO_BIT   = {1..0};
      REG_FOUR  = REGISTER(4,LOAD,,,,RESET,ENABLE);

   port
      reset,x,clk        : input of EVENT;
      reg                : input of TWO_BIT;

   var
      a,c,d              : INTEGER;
      b                  : REG_FOUR;

}

TABLE example {

   OPS_BASED

   FIRST
   STATE: top
   {
      {
         CONDITION: (true);
         ACTIONS: ;
         NEXT_STATE: SUBTABLE Top_Table;
         EVENT: (call);
      },

      {
         CONDITION: (true);
         ACTIONS: ;
         NEXT_STATE: top;
         EVENT: (reset rising);
      }
   }
}

TABLE Top_Table {

   OPS_BASED

   FIRST
   STATE: h
   {
      {
         CONDITION: (true);
         ACTIONS: ;
         NEXT_STATE: SUBTABLE H_Table;
         EVENT: (call);
      },

      {
         CONDITION: (true);
         ACTIONS: ;
         NEXT_STATE: c;
```

```
            EVENT: (x falling);
         }
      },

      STATE: c
      {
         {
            CONDITION: (true);
            ACTIONS: ;
            NEXT_STATE: SUBTABLE C_Table;
            EVENT: (call);
         },

         {
            CONDITION: (true);
            ACTIONS: ;
            NEXT_STATE: h;
            EVENT: (timeout 150 ns);
         }
      }
   }

   TABLE H_Table {

      CONCURRENT {

         SUBTABLE A_Table,
         SUBTABLE B_Table
      }
   }

   TABLE A_Table {

      OPS_BASED

      FIRST
      STATE: 1
      {
         {
            CONDITION: (y=='1');
            ACTIONS: a = a + 1;
            NEXT_STATE: 1;
            EVENT: (clk falling);
         },
         {
            CONDITION: (else);
            ACTIONS: ;
            NEXT_STATE: 2;
            EVENT: (clk falling);
         }
      },

      STATE: 2
      {
         {
            CONDITION: (true);
            ACTIONS: a = a + 2;
            NEXT_STATE: 1;
            EVENT: (clk falling);
         }
      }
   }

   TABLE B_Table {

      OPS_BASED

      FIRST
```

```
      STATE: 1
      {
         {
            CONDITION: (true);
            ACTIONS: b|(delay 1 ns) = b + "0010";
            NEXT_STATE: 2;
            EVENT: (clk falling);
         }
      },

      STATE: 2
      {
         {
            CONDITION: (true);
            ACTIONS: ;
            NEXT_STATE: 2 OF TABLE C_Table;
            EVENT: (clk falling);
         }
      }
   }

   TABLE C_Table {

      OPS_BASED

      FIRST
      STATE: 1
      {
         UC_ACTIONS: c = c + 1;
         {
            CONDITION: (reg=="00" || reg=="01");
            ACTIONS: ;
            NEXT_STATE: 1;
            EVENT: (timeout 5 ns);
         },
         {
            CONDITION: (reg=="10");
            ACTIONS: ;
            NEXT_STATE: SUBTABLE D_Table;
            EVENT: (call);
         },
         {
            CONDITION: (else);
            ACTIONS: ;
            NEXT_STATE: 2;
            EVENT: (timeout 2 ns);
         }
      },

      STATE: 2
      {
         {
            CONDITION: (true);
            ACTIONS: ;
            NEXT_STATE: SUBTABLE E_Table;
            EVENT: (x rising);
         }
      }
   }

   TABLE D_Table {

      OPS_BASED

      FIRST
      STATE: 1
      {
         {
```

```
         CONDITION: (d < 10);
         ACTIONS: d|(delay 1 ns) = d + 1;
         NEXT_STATE: 1;
         EVENT: (clk falling);
      },
      {
         CONDITION: (else);
         ACTIONS: ;
         NEXT_STATE: TABLE C_Table;
         EVENT: (clk falling);
      }
   }
}

TABLE E_Table {

   OPS_BASED

   FIRST
   STATE: 1
   {
      {
         CONDITION: (d > 0);
         ACTIONS: d = d - 1;
         NEXT_STATE: 1;
         EVENT: (clk falling);
      },
      {
         CONDITION: (else);
         ACTIONS: ;
         NEXT_STATE: TABLE C_Table;
         EVENT: (clk falling);
      }
   }
}
```

A.2 Technical Report Example: VHDL

```
LIBRARY PACK;
USE PACK.DEFS.ALL;

ENTITY example_e IS
  PORT (reset: in bit;
        x: in bit;
        y: in bit;
        clk: in bit;
        reg: in bit_vector(1 downto 0));
END example_e;

ARCHITECTURE example_a OF example_e IS
  TYPE TABLE_top_table IS (top_table_h,top_table_bif_0tree_1);
  TYPE top_table_RES IS ARRAY(NATURAL RANGE <>) OF TABLE_top_table;
  FUNCTION top_table_RES_FUN (INPUT: top_table_RES) RETURN TABLE_top_table IS
  BEGIN
    RETURN INPUT(0);
  END top_table_RES_FUN;
  SIGNAL top_table: top_table_RES_FUN TABLE_top_table REGISTER := top_table_h;

  TYPE TABLE_a_table IS (a_table_1,a_table_2);
  TYPE a_table_RES IS ARRAY(NATURAL RANGE <>) OF TABLE_a_table;
  FUNCTION a_table_RES_FUN (INPUT: a_table_RES) RETURN TABLE_a_table IS
  BEGIN
    RETURN INPUT(0);
  END a_table_RES_FUN;
  SIGNAL a_table: a_table_RES_FUN TABLE_a_table REGISTER := a_table_1;

  TYPE TABLE_b_table IS (b_table_1,b_table_2);
  TYPE b_table_RES IS ARRAY(NATURAL RANGE <>) OF TABLE_b_table;
  FUNCTION b_table_RES_FUN (INPUT: b_table_RES) RETURN TABLE_b_table IS
  BEGIN
    RETURN INPUT(0);
  END b_table_RES_FUN;
  SIGNAL b_table: b_table_RES_FUN TABLE_b_table REGISTER := b_table_1;

  TYPE TABLE_bif_0tree_1 IS (bif_0tree_1_c_table,bif_0tree_1_d_table,bif_0tree_1_e_table);
  TYPE bif_0tree_1_RES IS ARRAY(NATURAL RANGE <>) OF TABLE_bif_0tree_1;
  FUNCTION bif_0tree_1_RES_FUN (INPUT: bif_0tree_1_RES) RETURN TABLE_bif_0tree_1 IS
  BEGIN
    RETURN INPUT(0);
  END bif_0tree_1_RES_FUN;
  SIGNAL bif_0tree_1: bif_0tree_1_RES_FUN TABLE_bif_0tree_1 REGISTER := bif_0tree_1_c_table;

  TYPE TABLE_c_table IS (c_table_1,c_table_2);
  TYPE c_table_RES IS ARRAY(NATURAL RANGE <>) OF TABLE_c_table;
  FUNCTION c_table_RES_FUN (INPUT: c_table_RES) RETURN TABLE_c_table IS
  BEGIN
    RETURN INPUT(0);
  END c_table_RES_FUN;
  SIGNAL c_table: c_table_RES_FUN TABLE_c_table REGISTER := c_table_1;

  TYPE a_RES IS ARRAY(NATURAL RANGE <>) OF integer;
  FUNCTION a_RES_FUN (INPUT: a_RES) RETURN integer IS
  BEGIN
    RETURN INPUT(0);
  END a_RES_FUN;
  SIGNAL a: a_RES_FUN integer REGISTER := 0;

  TYPE d_RES IS ARRAY(NATURAL RANGE <>) OF integer;
  FUNCTION d_RES_FUN (INPUT: d_RES) RETURN integer IS
  BEGIN
    RETURN INPUT(0);
  END d_RES_FUN;
  SIGNAL d: d_RES_FUN integer REGISTER := 0;
```

```
   signal b: bit_vector(3 downto 0) := b"0000";
   signal c: integer := 0;

   signal bif_self_lb_0: integer := 0;
   signal bif_self_lb_4: integer := 0;
   signal bif_self_lb_8: integer := 0;
   signal bif_self_lb_10: integer := 0;
   signal bif_self_lb_11: integer := 0;

   signal bif_timeout_lb_2: bit := '0';
   signal bif_timeout_lb_8: bit := '0';

 BEGIN
 LB_0: BLOCK (reset='1' and not reset'stable(0 ns))
 BEGIN
   top_table <= guarded top_table_h;
   a_table <= guarded a_table_1;
   b_table <= guarded b_table_1;
   bif_0tree_1 <= guarded bif_0tree_1;
   c_table <= guarded c_table;
   bif_self_lb_0 <= guarded bif_self_lb_0 + 1;
 END BLOCK LB_0;

 LB_1: BLOCK (top_table=top_table_h and (x='0' and not x'stable(0 ns)))
 BEGIN
   top_table <= guarded top_table_bif_0tree_1;
   a_table <= guarded a_table;
   b_table <= guarded b_table;
   bif_0tree_1 <= guarded bif_0tree_1_c_table;
   c_table <= guarded c_table_1;
 END BLOCK LB_1;

 LB_2: BLOCK (top_table=top_table_bif_0tree_1)
 BEGIN
   LB_2_0: PROCESS
   BEGIN
     IF GUARD THEN
       bif_timeout_lb_2 <= '0','1' after 150 ns;
     ELSE
     END IF;
   WAIT ON GUARD;
   END PROCESS LB_2_0;
 END BLOCK LB_2;

 LB_3: BLOCK (top_table=top_table_bif_0tree_1 and (bif_timeout_lb_2='1' and not bif_timeout_lb_2'stable(0 ns)))
 BEGIN
   top_table <= guarded top_table_h;
   a_table <= guarded a_table_1;
   b_table <= guarded b_table_1;
   bif_0tree_1 <= guarded bif_0tree_1;
   c_table <= guarded c_table;
 END BLOCK LB_3;

 LB_4: BLOCK (top_table=top_table_h and a_table=a_table_1)
   signal bif_case: integer := 0;
 BEGIN
   LB_4_1: PROCESS
   BEGIN
     IF GUARD THEN
       if y='1' then
         a <= a + 1;
         bif_case <= 1;
       else
         bif_case <= 2;
       end if;
     ELSE
       a <= null;
```

```
      END IF;
    WAIT ON GUARD,bif_self_lb_0,bif_self_lb_4;
    END PROCESS LB_4_1;
    LB_4_2: PROCESS
    BEGIN
      IF GUARD THEN
        if not (reset='1' and not reset'stable) then
          if bif_case=1 then
            if clk='0' and not clk'stable then
              bif_self_lb_4 <= bif_self_lb_4 + 1;
            end if;
          elsif bif_case=2 then
            if clk='0' and not clk'stable then
              a_table <= a_table_2;
            end if;
          end if;
        end if;
      ELSE
        a_table <= null;
      END IF;
    WAIT ON GUARD,clk;
    END PROCESS LB_4_2;
  END BLOCK LB_4;

  LB_5: BLOCK (top_table=top_table_h and a_table=a_table_2)
  BEGIN
    LB_5_1: PROCESS
    BEGIN
      IF GUARD THEN
        a <= a + 2;
      ELSE
        a <= null;
      END IF;
    WAIT ON GUARD;
    END PROCESS LB_5_1;
    LB_5_2: PROCESS
    BEGIN
      IF GUARD THEN
        if clk='0' and not clk'stable then
          a_table <= a_table_1;
        end if;
      ELSE
        a_table <= null;
      END IF;
    WAIT ON GUARD,clk;
    END PROCESS LB_5_2;
  END BLOCK LB_5;

  LB_6: BLOCK (top_table=top_table_h and b_table=b_table_1)
  BEGIN
    LB_6_1: PROCESS
    BEGIN
      IF GUARD THEN
        b <= transport b + b"0010" after 1 ns;
      END IF;
    WAIT ON GUARD,bif_self_lb_0;
    END PROCESS LB_6_1;
    LB_6_2: PROCESS
    BEGIN
      IF GUARD THEN
        if not (reset='1' and not reset'stable) then
          if clk='0' and not clk'stable then
            b_table <= b_table_2;
          end if;
        end if;
      ELSE
        b_table <= null;
      END IF;
```

```
    IF GUARD THEN
      if bif_case=1 then
        if clk='0' and not clk'stable then
          bif_self_lb_11 <= bif_self_lb_11 + 1;
        end if;
      elsif bif_case=2 then
        if clk='0' and not clk'stable then
          bif_0tree_1 <= bif_0tree_1_c_table;
          c_table <= c_table_1;
        end if;
      end if;
    ELSE
      bif_0tree_1 <= null;
      c_table <= null;
    END IF;
  WAIT ON GUARD,clk;
  END PROCESS LB_11_2;
END BLOCK LB_11;

END example_a;

CONFIGURATION example_c OF example_e IS
   FOR example_a
   END FOR;
END example_c;
```

A.3 Supplementary Example 1: BIF

```
SYMBOL_TABLE {
   type
      Event     = {0};
   port
      RESET,X,Y       : input of Event;
}
TABLE Example_1 {
   OPS_BASED
   FIRST
   STATE: top
   {
      {
         CONDITION: (true);
         ACTIONS:;
         NEXT_STATE: TABLE Top_Table;
         EVENT: (call);
      },
      {
         CONDITION: (true);
         ACTIONS:;
         NEXT_STATE: top;
         EVENT: (RESET rising);
      }
   }
}  /* End TABLE Example_1 */
TABLE Top_Table {
   OPS_BASED
   FIRST
   STATE: A
   {
      {
         CONDITION: (true);
         ACTIONS:;
         NEXT_STATE: TABLE A_Table;
         EVENT: (call);
      }
   },
   STATE: B
   {
      {
         CONDITION: (true);
         ACTIONS:;
         NEXT_STATE: TABLE B_Table;
         EVENT: (call);
      }
   }
}  /* End TABLE Top_Table */
TABLE A_Table {
   OPS_BASED
   FIRST
   STATE: 1
   {
      {
         CONDITION: (true);
         ACTIONS:;
         NEXT_STATE: B OF TABLE Top_Table ;
         EVENT: (after 20 ns);
      },
      {
         CONDITION: (true);
         ACTIONS:;
         NEXT_STATE: 2;
         EVENT: (x rising);
      }
   },
```

```
    WAIT ON GUARD,clk;
    END PROCESS LB_6_2;
  END BLOCK LB_6;

  LB_7: BLOCK (top_table=top_table_h and b_table=b_table_2)
  BEGIN
    LB_7_2: PROCESS
    BEGIN
      IF GURD THEN
        if clk='0' and not clk'stable then
          top_table <= top_table_bif_0tree_1;
          bif_0tree_1 <= bif_0tree_1_c_table;
          c_table <= c_table_2;
        end if;
      ELSE
        top_table <= null;
        bif_0tree_1 <= null;
        c_table <= null;
      END IF;
    WAIT ON GUARD,clk;
    END PROCESS LB_7_2;
  END BLOCK LB_7;

  LB_8: BLOCK (top_table=top_table_bif_0tree_1 and bif_0tree_1=bif_0tree_1_c_table and c_table=c_table_1)
    signal bif_case: integer := 0;
  BEGIN
    LB_8_1: PROCESS
    BEGIN
      IF GUARD THEN
        c <= c + 1;
        if reg=b"00" or reg=b"01" then
          bif_timeout_lb_8 <= '0','1' after 5 ns;
          bif_case <= 1;
        elsif reg=b"10" then
          bif_0tree_1 <= bif_0tree_1_d_table;
          bif_case <= 2;
        else
          bif_timeout_lb_8 <= '0','1' after 2 ns;
          bif_case <= 3;
        end if;
      ELSE
        bif_0tree_1 <= null;
      END IF;
    WAIT ON GUARD,bif_self_lb_8;
    END PROCESS LB_8_1;
    LB_8_2: PROCESS
    BEGIN
      IF GUARD THEN
        if bif_case=1 then
          if bif_timeout_lb_8='1' and not bif_timeout_lb_8'stable then
            bif_self_lb_8 <= bif_self_lb_8 + 1;
          end if;
        elsif bif_case=3 then
          if bif_timeout_lb_8='1' and not bif_timeout_lb_8'stable then
            c_table <= c_table_2;
          end if;
        end if;
      ELSE
        c_table <= null;
      END IF;
    WAIT ON GUARD,bif_timeout_lb_8;
    END PROCESS LB_8_2;
  END BLOCK LB_8;

LB_9: BLOCK (top_table=top_table_bif_0tree_1 and bif_0tree_1=bif_0tree_1_c_table and c_table=c_table_2)
BEGIN
  LB_9_2: PROCESS
  BEGIN
```

```
      IF GUARD THEN
        if x='1' and not x'stable then
          bif_0tree_1 <= bif_0tree_1_e_table;
        end if;
      ELSE
        bif_0tree_1 <= null;
      END IF;
    WAIT ON GUARD,x;
    END PROCESS LB_9_2;
  END BLOCK LB_9;

  LB_10: BLOCK (top_table=top_table_bif_0tree_1 and bif_0tree_1=bif_0tree_1_d_table)
    signal bif_case: integer := 0;
  BEGIN
    LB_10_1: PROCESS
    BEGIN
      IF GUARD THEN
        if d < 10 then
          d <= transport d + 1 after 1 ns;
          bif_case <= 1;
        else
          bif_case <= 2;
        end if;
      ELSE
        d <= null;
      END IF;
    WAIT ON GUARD,bif_self_lb_10;
    END PROCESS LB_10_1;
    LB_10_2: PROCESS
    BEGIN
      IF GUARD THEN
        if bif_case=1 then
          if clk='0' and not clk'stable then
            bif_self_lb_10 <= bif_self_lb_10 + 1;
          end if;
        elsif bif_case=2 then
          if clk='0' and not clk'stable then
            bif_0tree_1 <= bif_0tree_1_c_table;
            c_table <= c_table_1;
          end if;
        end if;
      ELSE
        bif_0tree_1 <= null;
        c_table <= null;
      END IF;
    WAIT ON GUARD,clk;
    END PROCESS LB_10_2;
  END BLOCK LB_10;

  LB_11: BLOCK (top_table=top_table_bif_0tree_1 and bif_0tree_1=bif_0tree_1_e_table)
    signal bif_case: integer := 0;
  BEGIN
    LB_11_1: PROCESS
    BEGIN
      IF GUARD THEN
        if d > 0 then
          d <= d - 1;
          bif_case <= 1;
        else
          bif_case <= 2;
        end if;
      ELSE
        d <= null;
      END IF;
    WAIT ON GUARD,bif_self_lb_11;
    END PROCESS LB_11_1;
    LB_11_2: PROCESS
    BEGIN
```

```
   STATE: 2
   {
      {
         CONDITION: (true);
         ACTIONS:;
         NEXT_STATE: B OF TABLE Top_Table ;
         EVENT: (after 20 ns);
      },
      {
         CONDITION: (true);
         ACTIONS:;
         NEXT_STATE: 1;
         EVENT: (x falling);
      }
   }
}  /* End TABLE A_Table */
TABLE B_Table {
   OPS_BASED
   FIRST
   STATE: 1
   {
      {
         CONDITION: (true);
         ACTIONS:;
         NEXT_STATE: A OF TABLE Top_Table ;
         EVENT: (after 25 ns);
      },
      {
         CONDITION: (true);
         ACTIONS:;
         NEXT_STATE: 2;
         EVENT: (y rising);
      }
   },
   STATE: 2
   {
      {
         CONDITION: (true);
         ACTIONS:;
         NEXT_STATE: A OF TABLE Top_Table ;
         EVENT: (after 25 ns);
      },
      {
         CONDITION: (true);
         ACTIONS:;
         NEXT_STATE: 1;
         EVENT: (y falling);
      }
   }
}  /* End TABLE B_Table */
```

A.4 Supplementary Example 1: VHDL

```
ENTITY example_1_e IS
   PORT (reset,x,y: in bit);
END example_1_e;

ARCHITECTURE example_1_a OF example_1_e IS
   TYPE TABLE_top_table IS (top_table_a,top_table_b);
   TYPE top_table_RES IS ARRAY (NATURAL RANGE <>) OF TABLE_top_table;
   FUNCTION top_table_RES_FUN (INPUT: top_table_RES) RETURN TABLE_top_table IS
   BEGIN
     RETURN INPUT(0);
   END top_table_RES_FUN;
   SIGNAL top_table: top_table_RES_FUN TABLE_top_table REGISTER := top_table_a;

   TYPE TABLE_a_table IS (a_table_1,a_table_2);
   TYPE a_table_RES IS ARRAY (NATURAL RANGE <>) OF TABLE_a_table;
   FUNCTION a_table_RES_FUN (INPUT: a_table_RES) RETURN TABLE_a_table IS
   BEGIN
     RETURN INPUT(0);
   END a_table_RES_FUN;
   SIGNAL a_table: a_table_RES_FUN TABLE_a_table REGISTER := a_table_1;

   TYPE TABLE_b_table IS (b_table_1,b_table_2);
   TYPE b_table_RES IS ARRAY (NATURAL RANGE <>) OF TABLE_b_table;
   FUNCTION b_table_RES_FUN (INPUT: b_table_RES) RETURN TABLE_b_table IS
   BEGIN
     RETURN INPUT(0);
   END b_table_RES_FUN;
   SIGNAL b_table: b_table_RES_FUN TABLE_b_table REGISTER := b_table_1;

   signal bif_self_lb_0: integer := 0;
   signal bif_timeout_lb_1: bit := '0';
   signal bif_timeout_lb_2: bit := '0';
   signal bif_timeout_lb_3: bit := '0';
   signal bif_timeout_lb_4: bit := '0';

BEGIN
LB_0: BLOCK (reset='1' and not reset'stable(0 ns))
BEGIN
  top_table <= guarded top_table_a;
  a_table <= guarded a_table_1;
  b_table <= guarded b_table;
  bif_self_lb_0 <= guarded bif_self_lb_0 + 1;
END BLOCK LB_0;

LB_1: BLOCK (top_table=top_table_a and a_table=a_table_1)
BEGIN
  LB_1_1: PROCESS
  BEGIN
    IF GUARD THEN
      bif_timeout_lb_1 <= '0','1' after 20 ns;
    END IF;
  WAIT ON GUARD,bif_self_lb_0;
  END PROCESS LB_1_1;
  LB_1_2: PROCESS
  BEGIN
    IF GUARD THEN
      if not (reset='1' and not reset'stable) then
        if bif_timeout_lb_1='1' and not bif_timeout_lb_1'stable then
          top_table <= top_table_b;
          b_table <= b_table_1;
        elsif x='1' and not x'stable then
          a_table <= a_table_2;
        end if;
      end if;
    ELSE
```

```
      top_table <= null;
      a_table <= null;
      b_table <= null;
    END IF;
  WAIT ON GUARD,bif_timeout_lb_1,x;
  END PROCESS LB_1_2;
END BLOCK LB_1;

LB_2: BLOCK (top_table=top_table_a and a_table=a_table_2)
BEGIN
  LB_2_1: PROCESS
  BEGIN
    IF GUARD THEN
      bif_timeout_lb_2 <= '0','1' after 20 ns;
    END IF;
  WAIT ON GUARD;
  END PROCESS LB_2_1;
  LB_2_2: PROCESS
  BEGIN
    IF GUARD THEN
      if bif_timeout_lb_2='1' and not bif_timeout_lb_2'stable then
        top_table <= top_table_b;
        b_table <= b_table_1;
      elsif x='0' and not x'stable then
        a_table <= a_table_1;
      end if;
    ELSE
      top_table <= null;
      a_table <= null;
      b_table <= null;
    END IF;
  WAIT ON GUARD,bif_timeout_lb_2,x;
  END PROCESS LB_2_2;
END BLOCK LB_2;

LB_3: BLOCK (top_table=top_table_b and b_table=b_table_1)
BEGIN
  LB_3_1: PROCESS
  BEGIN
    IF GUARD THEN
      bif_timeout_lb_3 <= '0','1' after 25 ns;
    END IF;
  WAIT ON GUARD;
  END PROCESS LB_3_1;
   LB_3_2: PROCESS
  BEGIN
    IF GUARD THEN
      if bif_timeout_lb_3='1' and not bif_timeout_lb_3'stable then
        top_table <= top_table_a;
        a_table <= a_table_1;
      elsif y='1' and not y'stable then
        b_table <= b_table_2;
      end if;
    ELSE
      top_table <= null;
      a_table <= null;
      b_table <= null;
    END IF;
  WAIT ON GUARD,bif_timeout_lb_3,y;
  END PROCESS LB_3_2;
END BLOCK LB_3;

LB_4: BLOCK (top_table=top_table_b and b_table=b_table_2)
BEGIN
  LB_4_1: PROCESS
  BEGIN
    IF GUARD THEN
      bif_timeout_lb_4 <= '0','1' after 25 ns;
```

```
    END IF;
  WAIT ON GUARD;
  END PROCESS LB_4_1;
  LB_4_2: PROCESS
  BEGIN
    IF GUARD THEN
      if bif_timeout_lb_4='1' and not bif_timeout_lb_4'stable then
        top_table <= top_table_a;
        a_table <= a_table_1;
      elsif y='0' and not y'stable then
        b_table <= b_table_1;
      end if;
    ELSE
      top_table <= null;
      a_table <= null;
      b_table <= null;
    END IF;
  WAIT ON GUARD,bif_timeout_lb_4,y;
  END PROCESS LB_4_2;
END BLOCK LB_4;

END example_1_a;

CONFIGURATION example_1_c OF example_1_e IS
   FOR example_1_a
   END FOR;
END example_1_c;
```

A.5 Supplementary Example 2: BIF

```
SYMBOL_TABLE {

   type
      Event = {0};
      Cond  = {0};                /* Condition (a == 1) */

   port
      RESET,X,Y : input of Event;
      a         : input of Cond;

}

TABLE Example_2 {

   OPS_BASED

   FIRST
   STATE: top
   {
      {
         CONDITION: (true);
         ACTIONS:;
         NEXT_STATE: TABLE Top_Table;
         EVENT: (call);
      },

      {
         CONDITION: (true);
         ACTIONS:;
         NEXT_STATE: top;
         EVENT: (RESET rising);
      }
   }
}  /* End TABLE Example_2 */

TABLE Top_Table {

   OPS_BASED

   FIRST
   STATE: A
   {
      {
         CONDITION: (true);
         ACTIONS:;
         NEXT_STATE: TABLE A_Table;
         EVENT: (call);
      },

      {
         CONDITION: (true);
         ACTIONS:;
         NEXT_STATE: B;
         EVENT: (after 20 ns);
      }
   },

   STATE: B
   {
      {
         CONDITION: (true);
         ACTIONS:;
         NEXT_STATE: TABLE B_Table;
         EVENT: (call);
      },
```

```
      {
         CONDITION: (true);
         ACTIONS:;
         NEXT_STATE: A;
         EVENT: (after 25 ns);
      }
   }
} /* End TABLE Top_Table */

TABLE A_Table {

   OPS_BASED

   FIRST
   STATE: 1
   {
      {
         CONDITION: (a == '1');
         ACTIONS:;
         NEXT_STATE: 2;
         EVENT: (x rising);
      },

      {
         CONDITION: (else);
         ACTIONS:;
         NEXT_STATE: 1;
         EVENT: (x falling);
      }
   },

   STATE: 2
   {
      {
         CONDITION: (true);
         ACTIONS:;
         NEXT_STATE: 1;
         EVENT: (x falling);
      }
   }
} /* End TABLE A_Table */

TABLE B_Table {

   OPS_BASED

   FIRST
   STATE: 1
   {
      {
         CONDITION: (true);
         ACTIONS:;
         NEXT_STATE: 2;
         EVENT: (y rising);
      }
   },

   STATE: 2
   {
      {
         CONDITION: (true);
         ACTIONS:;
         NEXT_STATE: 1;
         EVENT: (y falling);
      }
   }
} /* End TABLE B_Table */
```

```
      IF GUARD THEN
        bif_timeout_lb_3 <= '0','1' after 25 ns;
      END IF;
    WAIT ON GUARD;
    END PROCESS LB_3_1;
  END BLOCK LB_3;

  LB_4: BLOCK (top_table=top_table_b and (bif_timeout_lb_3='1' and not bif_timeout_lb_3'stable(0 ns)))
  BEGIN
    top_table <= guarded top_table_a;
    a_table <= guarded a_table_1;
    b_table <= guarded b_table;
  END BLOCK LB_4;

  LB_5: BLOCK (top_table=top_table_a and a_table=a_table_1)
    signal bif_case: integer := 0;
  BEGIN
    LB_5_1: PROCESS
    BEGIN
      IF GUARD THEN
        if a='1' then
          bif_case <= 1;
        else
          bif_case <= 2;
        end if;
      END IF;
    WAIT ON GUARD,bif_self_lb_0,bif_self_lb_5;
    END PROCESS LB_5_1;
    LB_5_2: PROCESS
    BEGIN
      IF GUARD THEN
        if not (reset='1' and not reset'stable) then
          if bif_case=1 then
            if x='1' and not x'stable then
              a_table <= a_table_2;
            end if;
          elsif bif_case=2 then
            if x='0' and not x'stable then
              a_table <= a_table_1;
              bif_self_lb_5 <= bif_self_lb_5 + 1;
            end if;
          end if;
        end if;
      ELSE
        a_table <= null;
      END IF;
    WAIT ON GUARD,x;
    END PROCESS LB_5_2;
  END BLOCK LB_5;

  LB_6: BLOCK (top_table=top_table_a and a_table=a_table_2)
  BEGIN
    LB_6_2: PROCESS
    BEGIN
      IF GUARD THEN
        if x='0' and not x'stable then
          a_table <= a_table_1;
        end if;
      ELSE
        a_table <= null;
      END IF;
    WAIT ON GUARD,x;
    END PROCESS LB_6_2;
  END BLOCK LB_6;

  LB_7: BLOCK (top_table=top_table_b and b_table=b_table_1)
  BEGIN
    LB_7_2: PROCESS
```

A.6 Supplementary Example 2: VHDL

```
ENTITY example_2_e IS
   PORT (reset,x,y,a: in bit);
END example_2_e;

ARCHITECTURE example_2_a OF example_2_e IS
   TYPE TABLE_top_table IS (top_table_a,top_table_b);
   TYPE top_table_RES IS ARRAY (NATURAL RANGE <>) OF TABLE_top_table;
   FUNCTION top_table_RES_FUN (INPUT: top_table_RES) RETURN TABLE_top_table IS
   BEGIN
     RETURN INPUT(0);
   END top_table_RES_FUN;
   SIGNAL top_table: top_table_RES_FUN TABLE_top_table REGISTER := top_table_a;

   TYPE TABLE_a_table IS (a_table_1,a_table_2);
   TYPE a_table_RES IS ARRAY (NATURAL RANGE <>) OF TABLE_a_table;
   FUNCTION a_table_RES_FUN (INPUT: a_table_RES) RETURN TABLE_a_table IS
   BEGIN
     RETURN INPUT(0);
   END a_table_RES_FUN;
   SIGNAL a_table: a_table_RES_FUN TABLE_a_table REGISTER := a_table_1;

   TYPE TABLE_b_table IS (b_table_1,b_table_2);
   TYPE b_table_RES IS ARRAY (NATURAL RANGE <>) OF TABLE_b_table;
   FUNCTION b_table_RES_FUN (INPUT: b_table_RES) RETURN TABLE_b_table IS
   BEGIN
     RETURN INPUT(0);
   END b_table_RES_FUN;
   SIGNAL b_table: b_table_RES_FUN TABLE_b_table REGISTER := b_table_1;

   signal bif_self_lb_0: integer := 0;
   signal bif_self_lb_5: integer := 0;
   signal bif_timeout_lb_1: bit := '0';
   signal bif_timeout_lb_3: bit := '0';

BEGIN
LB_0: BLOCK (reset='1' and not reset'stable(0 ns))
BEGIN
  top_table <= guarded top_table_a;
  a_table <= guarded a_table_1;
  b_table <= guarded b_table;
  bif_self_lb_0 <= guarded bif_self_lb_0 + 1;
END BLOCK LB_0;

LB_1: BLOCK (top_table=top_table_a)
BEGIN
  LB_1_1: PROCESS
  BEGIN
    IF GUARD THEN
      bif_timeout_lb_1 <= '0','1' after 20 ns;
    END IF;
  WAIT ON GUARD,bif_self_lb_0;
  END PROCESS LB_1_1;
END BLOCK LB_1;

LB_2: BLOCK (top_table=top_table_a and (bif_timeout_lb_1='1' and not bif_timeout_lb_1'stable(0 ns)))
BEGIN
  top_table <= guarded top_table_b;
  a_table <= guarded a_table;
  b_table <= guarded b_table_1;
END BLOCK LB_2;

LB_3: BLOCK (top_table=top_table_b)
BEGIN
  LB_3_1: PROCESS
  BEGIN
```

```
  BEGIN
    IF GUARD THEN
      if y='1' and not y'stable then
        b_table <= b_table_2;
      end if;
    ELSE
      b_table <= null;
    END IF;
  WAIT ON GUARD,y;
  END PROCESS LB_7_2;
END BLOCK LB_7;

LB_8: BLOCK (top_table=top_table_b and b_table=b_table_2)
BEGIN
  LB_8_2: PROCESS
  BEGIN
    IF GUARD THEN
      if y='0' and not y'stable then
        b_table <= b_table_1;
      end if;
    ELSE
      b_table <= null;
    END IF;
  WAIT ON GUARD,y;
  END PROCESS LB_8_2;
END BLOCK LB_8;

END example_2_a;

CONFIGURATION example_2_c OF example_2_e IS
   FOR example_2_a
   END FOR;
END example_2_c;
```

A.7 Supplementary Example 3: BIF

```
/*
 * Complete Concurrency example from
 * BIF Technical Report Revision.
 */

SYMBOL_TABLE {

   type
      Event     = {0};

   port
      RESET,X   : input of Event;

}

TABLE Example_3 {

   OPS_BASED

   FIRST
   STATE: top
   {
      {
         CONDITION: (true);
         ACTIONS:;
         NEXT_STATE: TABLE Top_Table;
         EVENT: (call);
      },

      {
         CONDITION: (true);
         ACTIONS:;
         NEXT_STATE: top;
         EVENT: (RESET rising);
      }
   }
}  /* End TABLE Example_3 */

TABLE Top_Table {

   OPS_BASED

   FIRST
   STATE: H
   {
      {
         CONDITION: (true);
         ACTIONS:;
         NEXT_STATE: TABLE H_Table;
         EVENT: (call);
      },

      {
         CONDITION: (true);
         ACTIONS:;
         NEXT_STATE: C;
         EVENT: (X rising);
      }
   },

   STATE: C
   {
      {
         CONDITION: (true);
         ACTIONS:;
```

```
            NEXT_STATE: TABLE C_Table;
            EVENT: (call);
         }
      }
   }  /* End TABLE Top_Table */

   TABLE H_Table {

      CONCURRENT {

         TABLE A_Table,
         TABLE B_Table
      }
   }  /* End TABLE H_Table */

   TABLE A_Table {

      OPS_BASED

      FIRST
      STATE: 1
      {
         {
            CONDITION: (true);
            ACTIONS: reg = reg + 1;
            NEXT_STATE: 2;
            EVENT: (after 5 ns);
         }
      },

      STATE: 2
      {
         {
            CONDITION: (true);
            ACTIONS: reg = reg + 3;
            NEXT_STATE: H OF TABLE Top_Table;
            EVENT: (after 5 ns);
         }
      }
   }  /* End TABLE A_Table */

   TABLE B_Table {

      OPS_BASED

      FIRST
      STATE: 1
      {
         {
            CONDITION: (true);
            ACTIONS: index = 1;
            NEXT_STATE: 2;
            EVENT: (after 2 ns);
         }
      },

      STATE: 2
      {
         {
            CONDITION: (index < 3);
            ACTIONS: reg = reg + 2;
                     index = index + 1;
            NEXT_STATE: 2;
            EVENT: (after 2 ns);
         }
      }
   }  /* End of TABLE B_Table */
```

```
TABLE C_Table {

   OPS_BASED

   FIRST
   STATE: 1
   {
      {
         CONDITION: (true);
         ACTIONS: ;
         NEXT_STATE: 2;
         EVENT: (after 10 ns);
      }
   },

   STATE: 2
   {
      {
         CONDITION: (true);
         ACTIONS: ;
         NEXT_STATE: H OF TABLE Top_Table;
         EVENT: (X rising);
      }
   }
}  /* End TABLE C_Table */
```

A.8 Supplementary Example 3: VHDL

```
ENTITY example_3_e IS
   PORT (reset,x: in bit);
END example_3_e;

ARCHITECTURE example_3_a OF example_3_e IS
   TYPE TABLE_top_table IS (top_table_h,top_table_c);
   TYPE top_table_RES IS ARRAY(NATURAL RANGE <>) OF TABLE_top_table;
   FUNCTION top_table_RES_FUN (INPUT: top_table_RES) RETURN TABLE_top_table IS
   BEGIN
     RETURN INPUT(0);
   END top_table_RES_FUN;
   SIGNAL top_table: top_table_RES_FUN TABLE_top_table REGISTER := top_table_h;

   TYPE TABLE_a_table IS (a_table_1,a_table_2);
   TYPE a_table_RES IS ARRAY(NATURAL RANGE <>) OF TABLE_a_table;
   FUNCTION a_table_RES_FUN (INPUT: a_table_RES) RETURN TABLE_a_table IS
   BEGIN
     RETURN INPUT(0);
   END a_table_RES_FUN;
   SIGNAL a_table: a_table_RES_FUN TABLE_a_table REGISTER := a_table_1;

   TYPE TABLE_b_table IS (b_table_1,b_table_2);
   TYPE b_table_RES IS ARRAY(NATURAL RANGE <>) OF TABLE_b_table;
   FUNCTION b_table_RES_FUN (INPUT: b_table_RES) RETURN TABLE_b_table IS
   BEGIN
     RETURN INPUT(0);
   END b_table_RES_FUN;
   SIGNAL b_table: b_table_RES_FUN TABLE_b_table REGISTER := b_table_1;

   TYPE TABLE_c_table IS (c_table_1,c_table_2);
   TYPE c_table_RES IS ARRAY(NATURAL RANGE <>) OF TABLE_c_table;
   FUNCTION c_table_RES_FUN (INPUT: c_table_RES) RETURN TABLE_c_table IS
   BEGIN
     RETURN INPUT(0);
   END c_table_RES_FUN;
   SIGNAL c_table: c_table_RES_FUN TABLE_c_table REGISTER := c_table_1;

   TYPE a_reg_RES IS ARRAY(NATURAL RANGE <>) OF integer;
   FUNCTION a_reg_RES_FUN (INPUT: a_reg_RES) RETURN integer IS
   BEGIN
     RETURN INPUT(0);
   END a_reg_RES_FUN;
   SIGNAL a_reg: a_reg_RES_FUN integer REGISTER := 0;

   TYPE index_RES IS ARRAY(NATURAL RANGE <>) OF integer;
   FUNCTION index_RES_FUN (INPUT: index_RES) RETURN integer IS
   BEGIN
     RETURN INPUT(0);
   END index_RES_FUN;
   SIGNAL index: index_RES_FUN integer REGISTER := 0;

   signal b_reg: integer := 0;
   signal reg: integer := 100;
   signal bif_self_lb_0: integer := 0;
   signal bif_self_lb_3: integer := 0;
   signal bif_self_lb_5: integer := 0;
   signal bif_timeout_lb_2: bit := '0';
   signal bif_timeout_lb_3: bit := '0';
   signal bif_timeout_lb_4: bit := '0';
   signal bif_timeout_lb_5: bit := '0';
   signal bif_timeout_lb_6: bit := '0';

BEGIN
LB_0: BLOCK (reset='1' and not reset'stable)
BEGIN
```

```
   top_table <= guarded top_table_h;
   a_table <= guarded a_table_1;
   b_table <= guarded b_table_1;
   bif_self_lb_0 <= guarded bif_self_lb_0 + 1;
 END BLOCK LB_0;

 LB_1: BLOCK (top_table=top_table_h and (x='1' and not x'stable))
 BEGIN
   top_table <= guarded top_table_c;
   a_table <= guarded a_table;
   b_table <= guarded b_table;
   c_table <= guarded c_table_1;
 END BLOCK LB_1;

 LB_2: BLOCK (top_table=top_table_h and a_table=a_table_1)
 BEGIN
   LB_2_1: PROCESS BEGIN IF GUARD THEN
       a_reg <= reg + 1;
       bif_timeout_lb_2 <= '0','1' after 5 ns;
     ELSE
       a_reg <= null;
     END IF;
   WAIT ON GUARD,bif_self_lb_0;
   END PROCESS LB_2_1;
   LB_2_2: PROCESS
   BEGIN
     IF GUARD THEN
       if not (reset='1' and not reset'stable) then
         if bif_timeout_lb_2='1' and not bif_timeout_lb_2'stable then
           a_table <= a_table_2;
         end if;
       end if;
     ELSE
       a_table <= null;
     END IF;
   WAIT ON GUARD,bif_timeout_lb_2;
   END PROCESS LB_2_2;
 END BLOCK LB_2;

 LB_3: BLOCK (top_table=top_table_h and a_table=a_table_2)
 BEGIN
   LB_3_1: PROCESS
   BEGIN
     IF GUARD THEN
       a_reg <= reg + 3;
       bif_timeout_lb_3 <= '0','1' after 5 ns;
     ELSE
       a_reg <= null;
     END IF;
   WAIT ON GUARD;
   END PROCESS LB_3_1;
   LB_3_2: PROCESS
   BEGIN
     IF GUARD THEN
       if bif_timeout_lb_3='1' and not bif_timeout_lb_3'stable then
         a_table <= a_table_1;
         b_table <= b_table_1;
         bif_self_lb_3 <= bif_self_lb_3 + 1;
       end if;
     ELSE
       a_table <= null;
       b_table <= null;
     END IF;
   WAIT ON GUARD,bif_timeout_lb_3;
   END PROCESS LB_3_2;
 END BLOCK LB_3;

 LB_4: BLOCK (top_table=top_table_h and b_table=b_table_1)
```

```
BEGIN
  LB_4_1: PROCESS
  BEGIN
    IF GUARD THEN
      index <= 1;
      bif_timeout_lb_4 <= '0','1' after 2 ns;
    ELSE
      index <= null;
    END IF;
  WAIT ON GUARD,bif_self_lb_0;
  END PROCESS LB_4_1;
  LB_4_2: PROCESS
  BEGIN
      IF GUARD THEN
        if not (reset='1' and not reset'stable) then
          if bif_timeout_lb_4='1' and not bif_timeout_lb_4'stable then
            b_table <= b_table_2;
          end if;
        end if;
      ELSE
        b_table <= null;
      END IF;
  WAIT ON GUARD,bif_timeout_lb_4;
  END PROCESS LB_4_2;
END BLOCK LB_4;

LB_5: BLOCK (top_table=top_table_h and b_table=b_table_2)
BEGIN
  LB_5_1: PROCESS
  BEGIN
    IF GUARD THEN
      if index < 3 then
        b_reg <= reg + 2;
        bif_timeout_lb_5 <= '0','1' after 2 ns;
      end if;
    ELSE
      b_table <= null;
    END IF;
  WAIT ON GUARD,bif_self_lb_5;
  END PROCESS LB_5_1;
  LB_5_2: PROCESS
  BEGIN
      IF GUARD THEN
        if bif_timeout_lb_5='1' and not bif_timeout_lb_5'stable then
          b_table <= b_table_2;
          bif_self_lb_5 <= bif_self_lb_5 + 1;
          index <= index + 1;
        end if;
      ELSE
        index <= null;
        b_table <= null;
      END IF;
  WAIT ON GUARD,bif_timeout_lb_5;
  END PROCESS LB_5_2;
END BLOCK LB_5;

LB_6: BLOCK (top_table=top_table_c and c_table=c_table_1)
BEGIN
  LB_6_1: PROCESS
  BEGIN
    IF GUARD THEN
      bif_timeout_lb_6 <= '0','1' after 10 ns;
    END IF;
  WAIT ON GUARD;
  END PROCESS LB_6_1;
  LB_6_2: PROCESS
  BEGIN
      IF GUARD THEN
```

```
        if bif_timeout_lb_6='1' and not bif_timeout_lb_6'stable then
          c_table <= c_table_2;
        end if;
      ELSE
        c_table <= null;
      END IF;
  WAIT ON GUARD,bif_timeout_lb_6;
  END PROCESS LB_6_2;
END BLOCK LB_6;

LB_7: BLOCK (top_table=top_table_c and c_table=c_table_2)
BEGIN
  LB_7_2: PROCESS
  BEGIN
      IF GUARD THEN
        if x='1' and not x'stable then
          top_table <= top_table_h;
          a_table <= a_table_1;
          b_table <= b_table_1;
        end if;
      ELSE
        top_table <= null;
        a_table <= null;
        b_table <= null;
      END IF;
  WAIT ON GUARD,x;
  END PROCESS LB_7_2;
END BLOCK LB_7;

reg <=  a_reg when (not a_reg'stable) else
        b_reg when (not b_reg'stable) else
        reg;

END example_3_a;

CONFIGURATION example_3_c OF example_3_e IS
   FOR example_3_a
   END FOR;
END example_3_c;
```