Control/Data-flow Analysis for VHDL Semantic Extraction *

Yee-Wing Hsieh   Steven P. Levitan

Department of Electrical Engineering
University of Pittsburgh

Abstract

Model abstraction reduces the number of states necessary to perform formal verification while maintaining the functionality of the original model with respect to the specifications to be verified. However, in order to perform model abstraction, we must extract the semantics of the model itself. In this paper, we describe a method for extracting VHDL semantics for model abstraction to improve the performance of formal verification tools such as COSPAN.

1 Introduction

Validating a design using formal verification methods is inherently computationally intensive, and, as a result, only small designs can be verified. To handle large designs, model abstraction is necessary. Model abstraction takes a model and replaces it with a high-level description of non-deterministic automata that encapsulates the behavior of the model it replaces. Using this high-level abstract representation, model abstraction reduces the number of states necessary to perform formal verification and thus reduces the state space to be explored by formal verification tools.

In order to determine which signals or which parts of the original model to abstract, we must first extract the parts of the model that exhibit memory semantics. This is because the memory semantics of the model ultimately determine the state space explored by formal verification tools. In this paper, we describe a method for extracting VHDL semantics for model abstraction to improve the performance of formal verification tools such as COSPAN [HK93].

The remainder of the paper is organized as follows. First, we describe various aspects of VHDL semantics that our analysis tool extracts from the VHDL models. Afterwards, we present our control/data-flow analysis techniques with respect to various VHDL semantics issues. We then present our analysis algorithms for memory semantics extraction and pre-abstraction partitioning. Finally, we present experimental results and conclusions.

2 Semantics Extraction

The memory semantics of the model ultimately determine the state space explored by formal verification tools. Therefore, to perform model abstraction, we must first extract memory semantics from the model. Our semantic extraction technique is based on control/data-flow analysis of the VHDL model.

Control/data-flow analysis is a technique often used in compiler design for various code optimizations. For model abstraction, our control/data-flow analysis technique has two objectives: memory semantics extraction and pre-abstraction model partitioning.

Memory semantics extraction identifies both explicit and implicit memory semantics in the model. Explicit memory semantics analysis identifies signal or variable feedback paths. Feedback occurs when a signal or variable definition (e.g., assignment) has inputs that depend on its own value. A feedback path requires a memory element to maintain the value of the signal or variable so that it can be used as an input to evaluate the next output value.

Implicit memory semantics analysis identifies signals that are assigned in some control paths, but not all control paths. For example, a concurrent conditional signal assignment without a default ELSE clause has implicit memory semantics and requires a memory element to maintain the value of the signal.

Pre-abstraction model partitioning clusters portions of the model as specified by a group of definitions for a particular signal or variable. The resulting model partitioning serves as an initial partition for the model abstraction step. There are two cases where pre-abstraction model partitioning can be applied: variables and signals.

For variables, a model may be partitioned based on variable lifetime so that disjoint parts of the model can analyzed separately in the model abstraction step. For example, in the VHDL model shown in Figure 1,

[1] This research was supported, in part, by the National Science Foundation under Grant MIP-9102721

the definition of the variable x at d1 and its use in the definition for the variable y at d4 are clearly disjoint from the definitions of the variable x at d2 and d3 and its use within the WHILE loop at d3. In fact, the semantics of the model would be preserved if variable x is replaced with another variable in either of the two define-use chains.

```
process
    variable  x,y,a,b,c: integer
begin
  x := a + 1;              -- d1 --
  y := x * 2;              -- d4 --

  x := b;                  -- d2 --
  while (x < c) loop
     x := x + b;           -- d3 --
  end loop;
end process;
```

Figure 1: Variable lifetime Partitioning

For signals, a model may be partitioned based on BUS signal sources so that each source can be analyzed separately in the model abstraction step. Sometimes, several of these sources may be eliminated from a model if they are not related to the properties being verified. After model abstraction is performed, the abstractions for each of the driving sources are merged together to form an abstract model of the BUS.

3 Method

Our technique for performing semantic extraction can best be explained in terms of the rules for variable definition and use. Variable define-use analysis is a conservative strategy to analyze sequential code and statically determine the set of definitions reaching a particular point in a program [ASU86]. For semantic extraction, we modified the standard compiler technique with VHDL semantics for signals and variables.

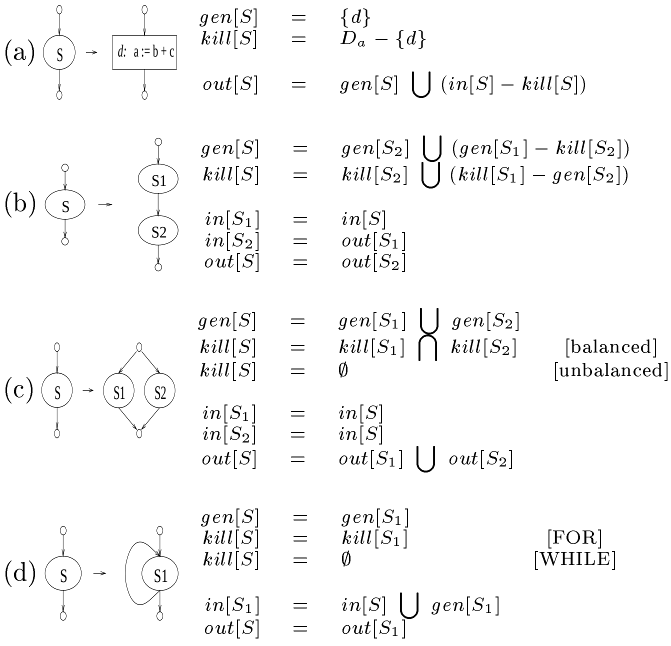
We define the following data-flow analysis terms:

d_S:	set of definitions at statement S
D_a:	set of definitions for variable a
$gen[S]$:	set of definitions generated that reach the end of S without following paths outside of S
$kill[S]$:	set of definitions overwritten that never reach the end of S
$in[S]$:	set of definitions that reach the beginning of S, taking into account control paths
$out[S]$:	set of definitions that reach the end of S, taking into account control paths

Each of these data-flow sets is calculated based on the structure of the model and VHDL semantics.

3.1 Variable Assignments

The VHDL variable assignment data-flow equations for single assignments, for compound statements, for if-statements and for loops are shown in Figures 2a, 2b, 2c and 2d, respectively.



$$\begin{aligned} gen[S] &= \{d\} \\ kill[S] &= D_a - \{d\} \\ out[S] &= gen[S] \bigcup (in[S] - kill[S]) \end{aligned}$$

$$\begin{aligned} gen[S] &= gen[S_2] \bigcup (gen[S_1] - kill[S_2]) \\ kill[S] &= kill[S_2] \bigcup (kill[S_1] - gen[S_2]) \\ in[S_1] &= in[S] \\ in[S_2] &= out[S_1] \\ out[S] &= out[S_2] \end{aligned}$$

$$\begin{aligned} gen[S] &= gen[S_1] \bigcup gen[S_2] & \\ kill[S] &= kill[S_1] \bigcap kill[S_2] & \text{[balanced]} \\ kill[S] &= \emptyset & \text{[unbalanced]} \\ in[S_1] &= in[S] & \\ in[S_2] &= in[S] & \\ out[S] &= out[S_1] \bigcup out[S_2] & \end{aligned}$$

$$\begin{aligned} gen[S] &= gen[S_1] & \\ kill[S] &= kill[S_1] & \text{[FOR]} \\ kill[S] &= \emptyset & \text{[WHILE]} \\ in[S_1] &= in[S] \bigcup gen[S_1] & \\ out[S] &= out[S_1] & \end{aligned}$$

Figure 2: Data-flow Equations for Variable Assignment Reaching Definitions

The variable assignment reaching definitions apply directly to subprograms (FUNCTIONS and PROCEDURE). However, a PROCESS statement is a concurrent statement, so the list of sequential statements in the PROCESS block executes continuously (i.e., whenever an event occurs on any of the signals in the sensitivity list). The flow control of this repeated behavior is similar to a loop structure, so the $in[S]$ and $out[S]$ sets for a PROCESS block are modified accordingly. Specifically, all definitions reached by the beginning of the PROCESS block ($in[S_1]$) consist of all the definitions reached by the beginning of the PROCESS statement ($in[S]$) plus all the definitions generated by the PROCESS block ($gen[S_1]$). The reaching definitions for the PROCESS statement (and other concurrent statements) are presented in Section 3.3.

3.2 Sequential Signal Assignments

According to VHDL semantics, sequential signal assignments update their values at the end of a PROCESS block or at the beginning of a WAIT statement. To handle this delayed update effect, we define the $post\_gen[S], post\_kill[S], post\_in[S]$ and *post_out*[S] sets be the posted reaching definitions to be updated later. These sequential signal assignments reaching definitions correspond to and are evaluated exactly the same way as the *gen*[S], *kill*[S], *in*[S] and *out*[S] sets for variable assignments. At the end of a PROCESS block or at the beginning of a WAIT statement, these posted reaching definitions are integrated into corresponding *gen*[S], *kill*[S], *in*[S] and *out*[S] sets. The data-flow equations for updating the posted reaching definitions are listed below.

$$\begin{aligned} gen[S_2] &= post\_gen[S_1] \bigcup \\ &\quad (gen[S_1] - post\_kill[S_1]) \qquad (1) \\ kill[S_2] &= post\_kill[S_1] \bigcup \\ &\quad (kill[S_1] - post\_gen[S_1]) \qquad (2) \\ out[S_2] &= post\_gen[S_1] \bigcup \\ &\quad (in[S_1] - post\_kill[S_1]) \qquad (3) \end{aligned}$$

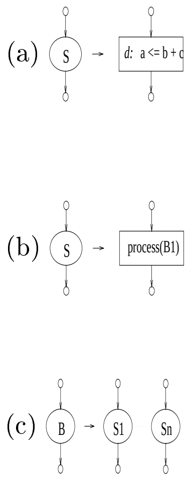
3.3 Concurrent Signal Assignments

Concurrent statements may consist of concurrent signal assignments such as conditional signal assignments or selected signal assignments, component instantiation and process statements. All concurrent statements, by definition, run in parallel, so the definitions generated by a concurrent statement can not be killed by definitions generated by any other concurrent statement. As a result, the *in*[S] and *out*[S] sets are the sum of all definitions generated by all concurrent statements. This is true in all cases except for the process statement where a definition for a signal (e.g., a sequential assignment) may kill other definitions within that process block hierarchy.

To capture the behavior of concurrent semantics, we define *gen*[B] and *kill*[B] to be the set of definitions generated and killed by the concurrent block B, respectively. By the definition of concurrency, *gen*[B] is also the *in*[S] and *out*[S] sets for each of the concurrent statements in the block. The data-flow equations for single concurrent signal assignments, for process statements and for multiple concurrent statements, are shown in Figure 3a, Figure 3b and Figure 3c, respectively.

3.4 Component Instantiations and Subprogram Calls

The reaching definitions (*gen*[S], *kill*[S], *in*[S], *out*[S]) for the component instantiations and subprogram calls are calculated just like a signal or variable assignment depending on the object of each output parameter. For example, if an actual parameter (variable x) corresponds to a formal parameter (variable y) of a subprogram and if the formal parameter is of the mode OUT or INOUT, then a definition for the actual parameter (variable x) is generated by the subprogram call. Since definitions may be generated by a component instantiation or by a subprogram call, then definitions may be killed in the same way as well. From the *gen*[S] and *kill*[S] sets, the *in*[S] and *out*[S] sets are calculated as if the component instantiation or the subprogram call is a signal or variable assignment.



$$\begin{aligned} gen[S] &= \{d\} \\ kill[S] &= \emptyset \\ in[S] &= gen[B] \\ out[S] &= gen[B] \end{aligned}$$

$$\begin{aligned} gen[S] &= gen[B_1] \\ kill[S] &= kill[B_1] \\ in[S] &= gen[B] \\ out[S] &= gen[B] \end{aligned}$$

$$\begin{aligned} gen[B] &= \textstyle\bigcup_{\forall S \in B} \; gen[S] \\ kill[B] &= \textstyle\bigcup_{\forall S \in B} \; kill[S] \\ out[B] &= gen[B] \end{aligned}$$

Figure 3: Data-flow Equations for Concurrent Signal Assignment Reaching Definitions

4 Analysis Algorithms

Our semantic extraction method consists of several analysis algorithms built upon the control/data-flow analysis technique presented earlier. Each algorithms analyzes the statements in the VHDL model by traversing a hierarchical representation of the model generated by our VHDL compiler front-end. Each of the analysis algorithm operates on a common data-structure. The results generated from each analysis are stored with each statement.

The reaching definitions are calculated first. Since the *in*[S] and *out*[S] sets are defined in terms of the *gen*[S] and *kill*[S] sets, the *gen*[S] and *kill*[S] sets are calculated first for the entire model, while the *in*[S] and *out*[S] sets are calculated in a second pass. These algorithms were outlined in Section 3.

The input dependency of each statement is evaluated next, using the reaching definitions calculated earlier. Explicit memory semantics are extracted with feedback analysis by examining the input dependency of each assignment. Implicit memory semantics are extracted with control path analysis and input signal sensitivity analysis. Control path analysis examines signal assignments to determine if a path exists where an output signal is not assigned. Input signal sensitivity analysis examines input dependency of a sequential signal assignment and checks if all input signals for the assignment are sensitized. Following memory semantics extraction, pre-abstraction partitioning is performed. Variable lifetime partitioning examines the input dependency of the assignments and control expressions to form disjoint groups of definitions for variables. BUS signal partitioning examines the output signals of each concurrent statement to identify the driving sources of BUS signals. To summarize, the pseudo-code for the control/data-flow analysis algorithm is simply:

```
control/data-flow analysis  {
    gen_kill_sets();
    in_out_sets();
    input_dependency();
    feedback_check();
    control_path();
    input_signal_sensitivity_check();
    variable_lifetime_partitioning();
    bus_signal_partitioning();
}
```

With an understanding of the overall VHDL semantics extraction method, we present each of these analyses in the order of algorithm execution.

4.1 Input Dependency

A definition in a model has two types of dependencies. First, a definition has *data dependency* from the input signals or variables in the assignment expression. Second, a definition has *control dependency* from the input signals or variables in the control expressions of the entire control path. Therefore, the input dependency of a definition is the union of data dependencies and control dependencies. We define the following terms:

$\mathit{data\_depend}[d_S]$: data-flow dependencies for the definition at statement S

$\mathit{ctrl\_depend}[B]$: control-flow dependencies for the statement block B

$\mathit{input\_depend}[d_S]$: control/data-flow dependencies for the definition at statement S

The term $\mathit{data\_depend}[d_S]$ is defined as the assignment expression dependency for the definition at statement S. The term $\mathit{ctrl\_depend}[B]$ is defined as the control expression dependency in the entire control path to the statement block B. The term $\mathit{input\_depend}[d_S]$ is defined as the control and data dependency for the definition at statement S.

Due to the recursive nature of the dependencies, an iterative technique is required to calculate all three dependencies. Equation(4) through Equation(9) summarize the iterative algorithm.

$$\mathit{data\_depend}_0[d_S] = in[S] \bigcap \Big(\bigcup_{\forall input \, \in \, S} D_{input} \Big) \tag{4}$$

$$\mathit{ctrl\_depend}_0[B] = \Big(in[B] \bigcap \Big(\bigcup_{\forall input \, \in \, ctrl\_expr} D_{input}\Big)\Big) \bigcup \mathit{ctrl\_depend}_0[P] \tag{5}$$

$$\mathit{input\_depend}_0[d_S] = \mathit{data\_depend}_0[d_S] \bigcup \mathit{ctrl\_depend}_0[B] \tag{6}$$

$$\mathit{data\_depend}_i[d_S] = \Big(\bigcup_{\forall d_j \, \in \, data\_depend_{i-1}[d_S]} \mathit{data\_depend}_{i-1}[d_j]\Big) \tag{7}$$

$$\mathit{ctrl\_depend}_i[B] = \Big(\bigcup_{\forall d_j \, \in \, ctrl\_depend_{i-1}[B]} \mathit{ctrl\_depend}_{i-1}[d_j]\Big) \tag{8}$$

$$\mathit{input\_depend}_i[d_S] = \Big(\bigcup_{\forall d_j \, \in \, input\_depend_{i-1}[d_S]} \mathit{input\_depend}_{i-1}[d_j]\Big) \tag{9}$$

Equation(4), Equation(5) and Equation(6) represent the initial iteration for the three dependencies, and they capture the first level of dependency. In Equation(4), the union part of the equation collects all of the definitions for each input in the assignment expression. The intersection part of the equation keeps only those definitions that reach the beginning of each assignment (i.e., $in[S]$). Similarly, in Equation(5), the union part of the equation sums of all the definitions for each input in the control expression for this block, while the intersection part keeps only those definitions that reach the beginning of each control expression. The last union in Equation(5) captures the control path hierarchy by incorporating the control dependency from the parent block P [2]. In Equation(6), the input dependency for the definition at statement S is the sum of the data dependencies and control dependencies.

Equation(7), Equation(8) and Equation(9) represent the update iterations for the three dependencies, and they capture recursive dependencies. These three equations are all alike, and they state that the dependency in the current iteration (i) is the sum of the dependencies from each of the definitions in the dependency set calculated in the previous iteration ($i-1$).

[2] It is important to note that input dependency analysis is performed in a top-down manner. Therefore, it is sufficient to incorporate the control dependency from just the parent block P verses all the ancestors.

The iteration stops when there is no change in any of the dependency sets for all the definitions in the model. In theory, it will take at most n iterations where n is the depth of the longest recursive dependency.

The result of the input dependency analysis enables us to perform explicit memory feedback analysis, input signal sensitivity analysis and variable lifetime partitioning. We discuss these next.

4.2 Feedback Analysis

The feedback analysis determines if a definition has inputs that depend on itself. The term $feedback[d_S]$ is a boolean indicating whether the definition at statement S contains feedback. Equation(10) summarizes the feedback analysis. The equation states that the definition d_S contains feedback if the definition d_S itself is in the definition's input dependency set. Otherwise, d_S does not contain feedback.

$$feedback[d_S] = \begin{cases} 1 & \text{if} \quad d_S \in input\_depend[d_S] \\ 0 & \text{otherwise} \end{cases} \qquad (10)$$

To identify explicit memory semantics in a model, the algorithm performs the feedback analysis on every sequential and concurrent assignment.

4.3 Control Paths

In the VHDL language, both concurrent and sequential signal assignments could have implicit memory semantics. Specifically, for concurrent signal assignments, both selected signal assignments and conditional signal assignments may have implicit memory semantics depending on whether all the conditionals are balanced or fully specified. For the conditional signal assignment, a simple check for the absence of the ELSE clause without a conditional expression would determine whether the conditional signal assignment has implicit memory. For the selected signal assignment, the balanced conditionals check consists of verifying if all the cases for the selected expression are fully enumerated and if the default case is specified. If neither condition is satisfied, then the selected signal assignment has implicit memory semantics.

For sequential signal assignments, the implicit memory analysis is a bit more complicated, due to the fact that a sequential signal assignment does not update its value until the end of a PROCESS block or at the beginning of a WAIT statement, and the fact that sequential assignments may overwrite one another. As a result, a simple balanced check of sequential signal assignments in conditional structures such as IF statements or CASE statements will not suffice. Instead, to determine whether an output signal has implicit memory semantics, every control path must be examined to check for a control path where the output signal is not assigned. To solve this implicit memory problem, we define the possible memory set ($pm[B]$) to be the set of output signals *not* assigned in the current sequential block hierarchy B.

$pm[B]$: set of signals not assigned
in statement block B

The $pm[B]$ set in each statement block represents signals not assigned within the segment of control path from the beginning to the end of each statement block. By combining the $pm[B]$ sets from each control path hierarchically, using the data-flow rules presented here, we can determine which signals are *not* assigned in every control path.

In the control path analysis algorithm, the $pm[B]$ set in each sequential statement block is initialized to include all output signals present in the entire PROCESS block or SUBPROGRAM block hierarchy. Next, the algorithm examines each of the sequential blocks hierarchically and modifies the possible memory set $pm[B]$ according to the data-flow equations shown in Figure 4.

The data-flow equation in Figure 4a states that for each signal assignment in the block B, the output signal is removed from $pm[B]$. The data-flow equations for conditionals are shown in Figure 4b. For unbalanced conditionals, the $pm[B]$ set is not changed because a path exists where signals are not assigned. On the other hand, for balanced conditionals, we remove a signal from the $pm[B]$ set in the current block B if the output signal is assigned in each of the conditional blocks B_1 and B_2 . The data-flow equations for loops are shown in Figure 4c. For WHILE loops, the $pm[B]$ set is not changed because we can not statically determine for certain that a WHILE loop will always execute. However, a FOR loop will always execute at least once. Therefore, we remove a signal from the $pm[B]$ set in the current block B if the output signal is assigned in the block B_1 of the FOR loop.

At the end of the control path analysis, the $pm[B]$ set for the PROCESS block or SUBPROGRAM block is the set of signals not assigned in every control path. These signals have implicit memory semantics and require memory elements to maintain their value in those control paths where they are not assigned.

4.4 Input Signal Sensitivity Analysis

An output signal in a sequential signal assignment may have implicit memory due to unsensitized signals in its inputs. Input signal sensitivity analysis identifies this type of implicit memory by checking if every

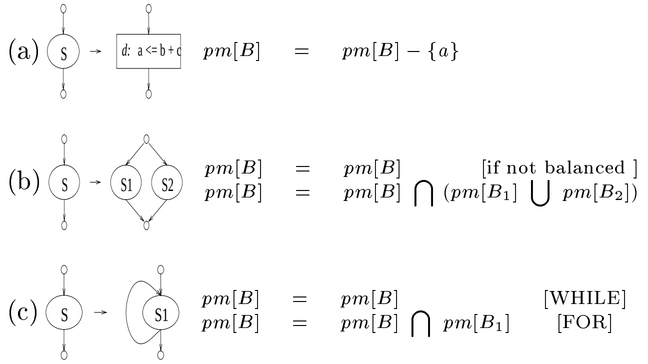

(a)
S
d: a<=b+c
$pm[B] = pm[B] - \{a\}$
(b)
S
S1
S2
$pm[B] = pm[B]$ [if not balanced]
$pm[B] = pm[B] \bigcap (pm[B_1] \bigcup pm[B_2])$
(c)
S
S1
$pm[B] = pm[B]$ [WHILE]
$pm[B] = pm[B] \bigcap pm[B_1]$ [FOR]

Figure 4: Data-flow Eqns. for Possible Memory Sets

signal in the input dependency of a sequential assignment is in the process sensitivity list. If any of the input signals in the input dependency are not in the process sensitivity list, then the output signal has implicit memory due to unsensitized input signals.

4.5 Variable Lifetime Partitioning

Variable lifetime partitioning divides each variable definition set into disjoint definition subsets based on where the definitions are used. Each of the disjoint definition subsets can later be analyzed as if they were defined as separate variables in the model abstraction step. The pseudo-code for the partitioning algorithm is listed below. The algorithm examines every input dependency ($input\_depend[d_S]$) in the model, and for every input dependency it extracts the set of definitions used by each input variable in the dependency set. The term $group_{input}$ represents the set of definitions used by a particular *input* variable in a definition at statement S. For each variable definition d_j, the term $Group[d_j]$ represents the disjoint subset of definitions for the same variable. By collecting the set of definitions used by each input variable, the result is a partitioning of the disjoint definitions in each variable definition set.

```
for each $depend[d_S]$ {
    for each $input$ in $d_S$ {
        $group_{input} = D_{input} \bigcap depend[d_S]$
        for each $d_j$ in $group_{input}$ {
            $Group[d_j] = Group[d_j] \bigcup group_{input}$
        }
    }
}
```

For the VHDL model shown in Figure 1, the definition set for variable x (D_x) is $\{d1,d2,d3\}$, and the definition set for variable y (D_y) is $\{d4\}$. The input dependency for the definition at $d4$ is $\{d1\}$, and therefore, $Group_{d1}$ is $\{d1\}$. However, the input dependency for the definition at $d3$ is $\{d2,d3\}$, and therefore, $Group_{d2}$ is $\{d2,d3\}$ and $Group_{d3}$ is $\{d2,d3\}$. As a result, the variable lifetime partitioning for variable x is $\{d1\}$ and $\{d2,d3\}$, shown in Figure 1 in Section 2.

4.6 BUS Signal Partitioning

As mentioned previously, a model may be abstracted by merging abstractions for each of the sources driving BUS the signal. To partition signals with multiple driving sources (i.e., BUS signals) for analysis in the model abstraction step, we need to identify any signal being assigned in multiple concurrent statements. The algorithm identifies the BUS signals by counting the number of times an output signal is assigned by concurrent statements in the model. Partitioning is formed by identifying the BUS signals and the concurrent statements driving the signals.

5 Results

We have implemented our VHDL semantics extraction tool using the control/data-flow analysis method presented above. To demonstrate how each of the analysis algorithms extract memory semantics, we present the results of semantic extraction for the set of VHDL models shown in Figure 5. We have kept the models in Figure 5 small in order to simplify the discussion of various VHDL semantics issues and the analysis algorithm that handles them. In all the models, identifiers w, x, y and z are signals and identifiers i, j, k and m are variables. For discussion purposes, each of the definitions d_S in the models is enumerated as comments after each signal or variable assignment.

The results of the semantic extraction performed on the models are shown in Figure 6. Column one of the table indicates the VHDL model of the semantic extraction experiment. Columns two and three indicate the definitions that have explicit memory semantics due to data dependency and control dependency, respectively. Columns four and five indicate signals that have implicit memory semantics due to unbalanced signal assignments and unsensitized input signals, respectively. Finally, the results of variable lifetime partitioning and BUS signal partitioning are shown in columns six and seven.

Returning to the semantic extraction experiments, we present our experimental results starting with the model shown in Figure 5a. In this model, the two guarded blocks, $b1$ and $b2$, have two guarded signal assignments on signal z, at d2 and d4. They are the only definitions affected by the guarded expressions. As a result, the guarded signal assignment at d2 has

```
b1: block (z = 1)
begin
   y <= a;              -- d1 --
   z <= guarded a;   -- d2 --
end block;

b2: block (x = 1)
begin
   x <= b;              -- d3 --
   z <= guarded b;   -- d4 --
end block;
```

(a)

```
process
   variable i,j,k: integer;
begin
   if (i < 1) then
      i := i + 1;        -- d1 --
   elsif (j < 1) then
      j := 2             -- d2 --
   else
      i := j + 1;        -- d3 --
   end if;
end process;
```

(b)

```
process
begin
   if (x < 1) then
      x <= x + 1;     -- d1 --
   elsif (y < 1) then
      y <= 2;         -- d2 --
   else
      x <= y + 1;     -- d3 --
   end if;
end process;
```

(c)

```
process
   variable i,j,k,m: integer
begin
   i := m;                   -- d5 --
   case i is
      when 2 =>
         k := j + 1;         -- d1 --
      when 3 =>
         k := i + 1;         -- d2 --
      when others =>
         if (k < 0) then
            i := j;          -- d3 --
            k := 0;          -- d4 --
         end if;
   end case;
end process;
```

(d)

```
process
   variable i,j,k: integer
begin
   case i is
      when 2 =>
         k := j + 1;         -- d1 --
      when 3 =>
         k := i + 1;         -- d2 --
      when others =>
         if (k < 0) then
            i := j;          -- d3 --
            k := 0;          -- d4 --
         end if;
   end case;
end process;
```

(e)

```
process (y)
begin
   case x is
      when 2 =>
         z <= y + 1;         -- d1 --
      when 3 =>
         z <= x + 1;         -- d2 --
      when others =>
         if (z < 0) then
            x <= y;          -- d3 --
            z <= 0;          -- d4 --
         end if;
   end case;
end process;
```

(f)

```
process (x,y,z)
begin
   z <= 0;                   -- d5 --
   case x is
      when 2 =>
         z <= y + 1;         -- d1 --
      when 3 =>
         z <= x + 1;         -- d2 --
      when others =>
         if (z < 0) then
            x <= y;          -- d3 --
            z <= 0;          -- d4 --
         end if;
   end case;
end process;
```

(g)

```
process (x,y)
   variable i: integer;
begin
   i := x;                   -- d1 --
   while (i < 10) loop
      i := i * 2;            -- d2 --
      w <= i;                -- d3 --
   end loop;
   i := y;                   -- d4 --
   for k in 1 to 10 loop
      i := i * 3;            -- d5 --
      z <= i;                -- d6 --
   end loop;
end process;
```

(h)

```
process (x,y)
   variable i: integer;
begin
   i := x;                   -- d1 --
   while (i < 10) loop
      i := i * 2;            -- d2 --
      w <= i;                -- d3 --
   end loop;
end process;

process (x,y)
   variable i: integer;
begin
   i := y;                   -- d4 --
   for k in 1 to 10 loop
      i := i * 3;            -- d5 --
      z <= i;                -- d6 --
   end loop;
end process;
```

(i)

Figure 5: Experiments

ex.	Explicit Memory		Implicit Memory		Paritioning	
	Data	Ctrl	Ctrl Path	Sens.	Lifetime	Bus
a		d2				{d2,d4}
b	d1	d1,d2,d3				
c	d1	d1,d2,d3	x,y			
d		d4				
e		d1,d2,d3,d4				
f		d1,d2,d3,d4	x,z	x,z		
g		d1,d2,d3,d4	x			
h	d2,d5	d2	w		{d1,d2} {d4,d5}	
i	d2,d5	d2	w			

Figure 6: Semantic Extraction Results

explicit feedback due to control dependency on the input signal z in the guarded expression. Since signal z has multiple signal drivers, we partition the signal between the two signal drivers.

The two models shown in Figures 5b and 5c have the same nested control structure. The only difference between the two models is that variables i and j (Figure 5b) are replaced by signals x and y (Figure 5c). They demonstrate the subtleties in variable and signal assignments. In the two models, both the *elsif* part and the *else* part of the if-statement depend on the inputs of the elsif-expression and if-expression. Specifically, for the two models, the definitions at d2 and d3 have control dependencies on variables i and j (signals x and y). Since process blocks have an implicit loop, the $in[S]$ sets at d1, d2 and d3 are each {d1,d2,d3}. As a result, there is an explicit feedback due to data dependency at d1, and there are explicit feedback paths due to control dependencies at d1, d2 and d3. For the model with signal assignments, a balanced conditional signal assignment check is performed. Both signals x and y (Figure 5b) are not assigned in all the control paths, thus both signals have implicit memory semantics. Since the process sensitivity list is not specified, all input signals within the process block hierarchy are implicitly sensitized. Therefore, the input signal sensitivity check is not necessary.

The four models shown in Figures 5d, 5e, 5f and 5g have the same control structure with a nested if-statement within a case-statement. They demonstrate the subtleties of initial variable and signal assignments. The two models shown in Figures 5d (and Figure 5e) are similar to the two models shown in Figures 5g (and Figure 5f), with variables i, j and k replaced by signals x, y, and z. The two models shown in Figure 5d (and Figure 5g) have an initial variable assignment (and signal assignment) at d5.

For the model shown in Figure 5d, definitions d1 and d2 have control dependencies on input variable i from the case-expression, and definitions d3 and d4 have control dependencies on input variable i from the case-expression and input variable k from the if-expression. The definition for variable i at d5 kills the definition for variable i at d3. Therefore, the $in[S]$ sets at d1, d2 and d3 are each {d1,d2,d4,d5}, while the $in[S]$ set at d4 is {d1,d2,d3,d4}. As a result, there is no explicit feedback due to data dependency. However, both definitions d3 and d4 have control dependencies on variable k, and the $in[S]$ sets at d3 and d4 include the definition d4. Consequently, the definition at d4 has explicit feedback due to the control dependencies on input variable k from the if-expression.

For the model shown in Figure 5e, the removal of the variable assignment at d5 results in definitions d1, d2, d3 and d4 to have control dependencies on variables i and k, even though definitions d1 and d2 are not defined within the if-statement. The reason is that variable i, defined at d3, depends on input variable k from the if-expression. Specifically, the $in[S]$ sets for d1 d2, d3 and d4 are each {d1,d2,d3,d4}. Therefore, definitions d1, d2, d3 and d4 have explicit feedback due to the control dependencies on variables i and k.

The model shown in Figure 5f has similar data dependency and control dependency as presented in the previous model. A balanced conditional signal assignment check performed on the model revealed that paths exist where signals x and z are not assigned. Therefore, signals x and z have implicit memory semantics due to unbalanced signal assignments in the conditionals. Furthermore, an input signal sensitivity check performed on the model revealed that both signals x and z depend on input signals x, y and z. However, only signal y is explicitly sensitized as specified in the process sensitivity list. Therefore, even if signals x and z have balanced signal assignments in the conditionals, both signals still have implicit memory semantics due to unsensitized input signals.

For the model shown in Figure 5g, the definition at d5 is the default assignment for signal z because signal assignments do not update their values until the end of a PROCESS block (or at the beginning of a WAIT statement). Due to the effects of a delayed update and the effects of the implicit loop within the process block, the $in[S]$ sets at d1, d2, d3, d4 and d5 are each {d1,d2,d3,d4,d5}. Consequently, the definitions containing explicit feedback due to control dependency include definitions d1, d2, d3 and d4. Due to the definition at d5, signal z is assigned in every control path even though it is not explicitly assigned in the else-part of the if-statement. Therefore, a balanced conditional signal assignment check revealed that only signal x has implicit memory semantics due to unbalanced conditional signal assignments. An input signal sensitivity check revealed that all input signals are included in the process sensitivity list. Hence, there are no implicit memory semantics due to unsensitized input signals.

The model shown in Figure 5h demonstrates variable lifetime partitioning. In this model, the variable i that is defined and used at d1, d2 and d3, is completely disjoint from the variable i that is defined and used at d4, d5 and d6. Therefore, a partitioning can be made between these two groups of variable lifetimes. The result of such a partitioning is shown in Figure 5i. Each definition from one model corresponds directly to a definition in the other model. The definitions at d2 and d5 in both models have explicit feedback due to data dependency. Also, the definition at d2 in both models has explicit feedback due to control dependency in the loop-expression. Furthermore, signal w in both models has implicit memory semantics due to an unbalanced conditional signal assignment of the while-loop. Similarly, signal z in both models does not have implicit memory semantics due to the balanced conditional signal assignment of the for-loop.

With the extracted explicit and implicit memory semantics, and with variable lifetime and BUS partitioning of the model, we obtain the abstract state space and initial partitioning necessary to perform model abstraction. The result is a model with abstractions that reduces the number of states necessary to perform formal verification and thus reduces the state space to be explored by formal verification tools such as COSPAN. We have shown [HL97] a 10 fold decrease in verification time as well as a 10^2 to 10^9 fold reduction in state space which allowed us to verify some models that were too large to verify without abstractions.

6 Conclusions

In this paper we have shown that in order to improve the performance of formal verification tools such as COSPAN, model abstraction is necessary. However, in order to perform model abstraction, we must obtain the semantics of the model itself. Therefore, we have implemented a VHDL semantic extraction tool that analyzes both concurrent and sequential VHDL models. The semantic extraction tool is based on data-flow analysis techniques in compiler designs for code optimization. The semantic extraction method identifies an abstract state space that is independent of synthesis optimizations and is therefore appropriate for model abstraction and verification before synthesis.

Acknowledgments The authors would like to thank Dr. Robert Kurshan of Lucent Technologies at Bell Laboratories for providing us with the COSPAN formal verification tool.

References

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley Publishing Company, 1986.

[HK93] Z. Har'El and R. Kurshan. *COSPAN User's Guide.* AT&T Bell Laboratories, Feb. 1993.

[HL97] Y. Hsieh and S. Levitan. Model abstraction for formal verification. Technical report TR-CE-201, Dept. of Elec. Eng., Univ. of Pittsburgh, Pittsburgh, PA, Jan. 1997.

[IEE94] IEEE, New York. *VHDL Language Reference Manual, IEEE Standard 1076-1993*, June 1994.