JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 14, 547-565 (1998)

Control / Data-flow Analysis for VHDL Semantic Extraction[1]

Yee-Wing Hsieh and Steven P. Levitan
Department of Electrical Engineering
University of Pittsburgh

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.

Keywords: VHDL semantics extraction, memory semantics extraction, control/data-flow analysis, model partitioning, model abstraction .

1. INTRODUCTION

Validating a design using formal verification methods is inherently computationally intensive; 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.

The traditional synthesis based technique in formal verification is to take a behavioral VHDL model, synthesize the design using this behavioral model, and then verify the resulting synthesis implementation of that design using various formal verification engines such as COSPAN [1], SMV [2] or Vis [3]. A pictorial representation of the synthesis based verification process is shown in Fig. 1(a).

The problem with the traditional synthesis based technique is that we are validating the synthesized implementation instead of the original model of a design. Also, there is a state space explosion problem due to registers used for temporary variables. More importantly, it is difficult to observe and trace the state of a variable when multiple registers may be allocated for a single variable or when a single

Received October 31, 1997; revised March 18, 1998.
Communicated by Jing-Yang Jou.
[1] This research was supported, in part, by the National Science Foundation under Grant MIP-9102721.

register may be allocated for multiple variables. As a result, it is difficult to perform abstraction because the behavioral semantics of the model are lost at the RTL level.

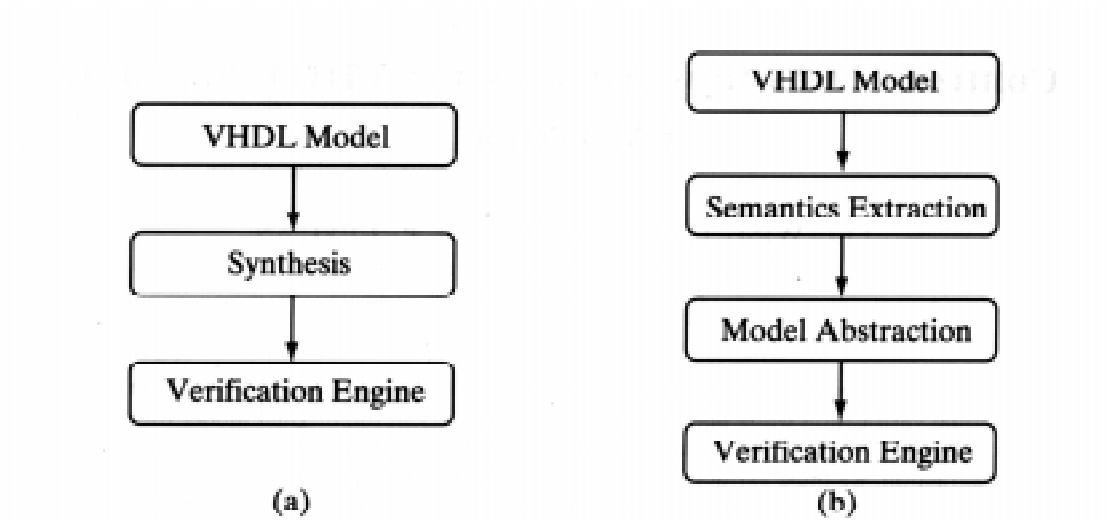

Fig. 1. Formal verification steps. (a) Traditional synthesis based. (b) Semantics based.

A better approach is to verify designs using an abstract state space. Our approach is a semantics based approach which extracts the semantics of the behavioral model, performs abstraction based on the semantics of the behavioral model, and then verifies the resulting abstract model. In our approach, semantic extraction identifies the abstract state space for model abstraction, instead of exploring the state space of the original, unreduced model. Using this abstract state space, model abstraction analysis identifies signals or parts of the original model for abstraction. A pictorial representation of the semantics based verification process is shown in Fig. 1(b).

The abstract state space identified by means of semantic extraction analysis is determined by 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 means of formal verification tools. In this paper, we describe a method for extracting VHDL [4] semantics for model abstraction to improve the perfor-mance of formal verification tools such as COSPAN. 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 means of 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 to 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 Fig. 2, 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 will be preserved if variable x is replaced with another variable in either of the two define-use chains.

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.

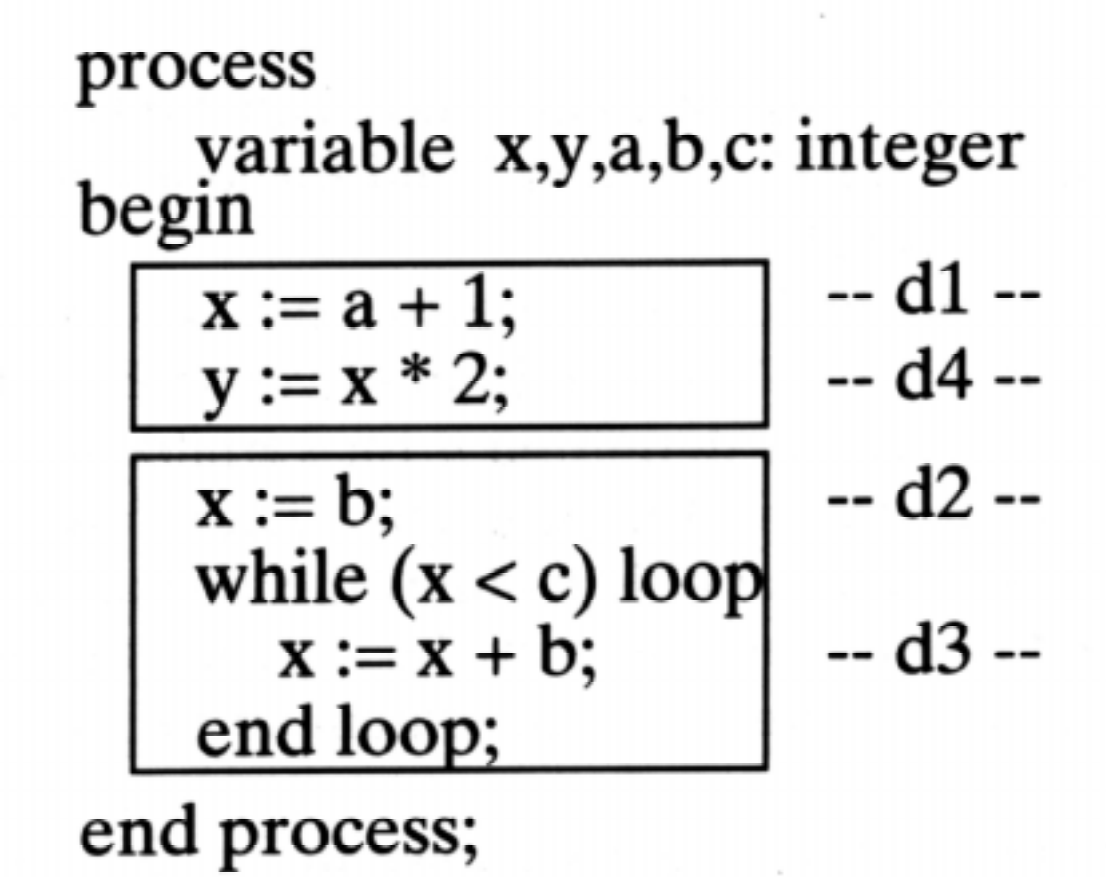

Fig. 2. Variable lifetime partitioning.

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 used to analyze sequential code and statically determine the set of definitions reaching a particular point in a program [5]. 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: the set of definitions in statement *S*,
D_a: the set of definitions for variable *a*,

gen[*S*]: the set of definitions generated that reach the end of *S* without following paths outside of *S*,
kill[*S*]: the set of definitions overwritten that never reach the end of S,
in[*S*]: the set of definitions that reach the beginning of *S*, taking into account

control paths,

out[*S*]: the 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 Fig. 3(a), 3(b), 3(c) and 3(d), respectively.

Observe the rule for single variable assignments, shown in Fig. 3(a). The variable assignment (*a*: = b + c) is a definition of variable *a*. Therefore, *gen*[*S*] is this definition *d*. However, this definition *d* for variable *a* kills (overwrites) all other definitions of variable *a* in the program. Therefore, *kill*[*S*] is the set of definitions in the program for variable a minus the current definition *d*. Finally, *out*[*S*] is the set of definitions generated by the statement (*gen*[*S*]) plus the set of definitions that reach the beginning of the statement (*in*[*S*]) that is not killed by the statement (*kill*[*S*]).

The rule for compound statements, shown in Fig. 3(b), is a bit more subtle. A definition is generated by a compound statement *S* if the definition is generated by statement S_2. It can also generated by statement S_1, provided that it is not killed by the definitions in statement S_2. Similarly, a definition is killed by a compound statement *S* if the definition is killed by statement S_2. It can also be killed by statement S_1, provided that it is not generated by the definitions in statement S_2. The equations for the *in*[*S*] and *out*[*S*] sets are more obvious. The set of definitions that reach the beginning of statement S1 (i.e., *in*[S_1]) is the set of definitions that reach the beginning of compound statement S (i.e., *in*[S_1]). The set of definitions that reach the beginning of statement S_2 (i.e., *in*[S_2]) is the set of definitions that reach the end of statement S_1 (i.e., out[S_1]). The set of definitions that reach the end of compound statement S (i.e., *out*[*S*]) is the set of definitions that reach the end of statement S_2 (i.e., *out*[S_2]).

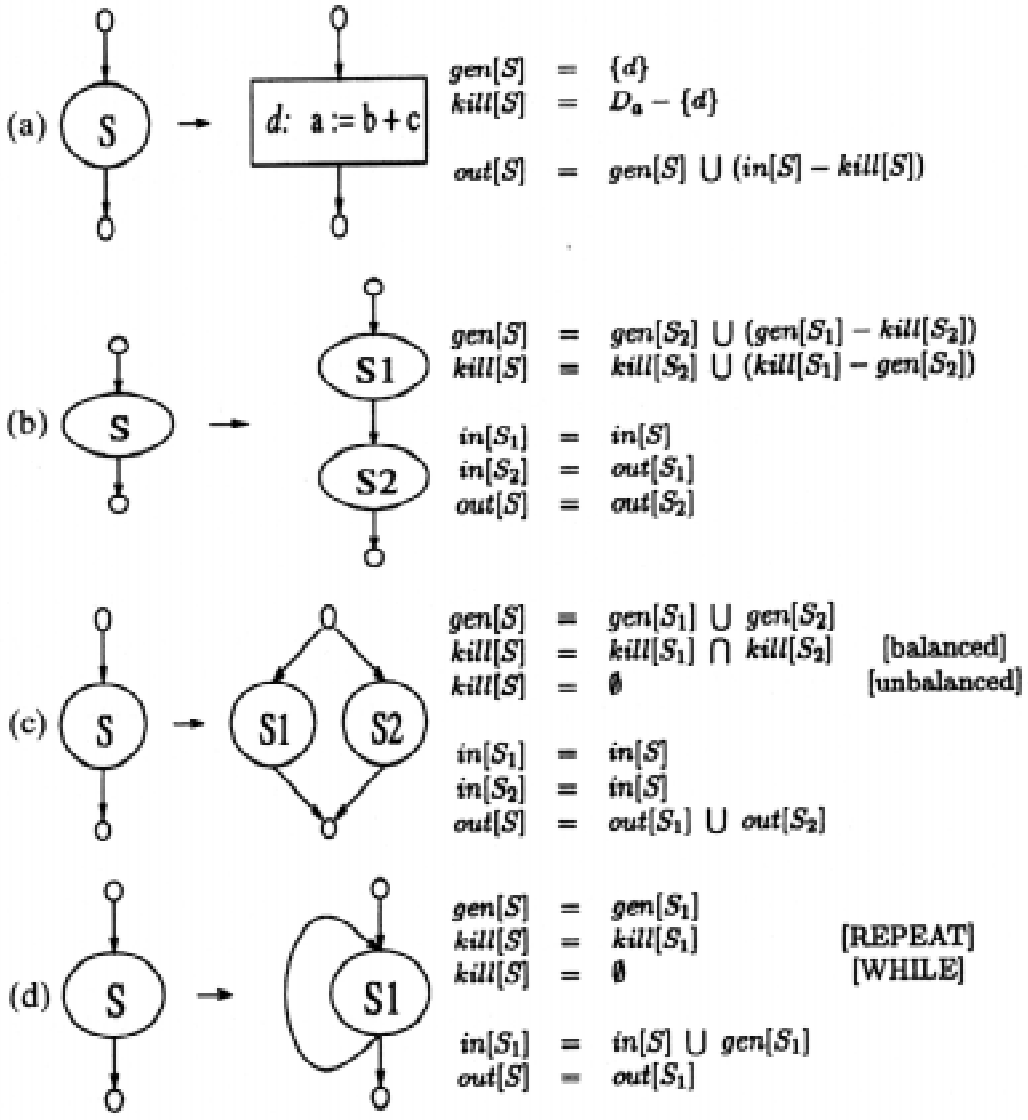

Fig. 3. Data-flow equations for variable assignment reaching definitions.

The rule for if-statements is shown in Fig. 3(c). Here, a definition is generated by if-statement S (i.e., *gen*[*S*]) if it is generated by either the statement block of the if-branch (S_1) or the statement block of the else-branch (S_2). On the other hand, a definition can be killed by if-statement S (i.e., kill[S]) if it is killed by the statement blocks of both the if-branch (S_1) and the else-branch (S_2). Consequently, if if-statement *S* does not have an else-clause, then no definitions can be killed by if-statement *S*[2]. For the same reason, this rule applies to case-statements that do not cover all possible cases. Obviously, the set of definitions that reach the beginning of if-statement S (i.e., *in*[*S*]) also reach the beginning of both statement blocks S_1 and S_2

[2] The rule for if-statements stated here is different from the one mentioned in *Compilers*, but it is a consistent derivation.

(i.e., *in*[S_1] and *in*[S_2]). The set of definitions that reach the end of if-statement *S* (i.e., *out*[*S*]) is the sum of definitions that reach the end of statement blocks S_1 and S_2 (i.e., *out*[S_1] and *out*[S_2]).

Observe the rule for loops, shown in Fig. 3(d). A definition is generated by a loop S (i.e., *gen*[*S*]) if it is generated by the loop statement block (S_1). On the other hand, a definition can be killed by loop *S* if the loop is executed at least once. Otherwise, no definition can be killed by loop *S*[3]. Since we can not statically analyze whether or not a while-loop will execute at least once, no definitions can be killed by a while-loop. Similarly, a for-loop may or may not execute at least once depending whether the loop parameter specification is a null discrete range. Therefore, a definition can be killed by loop S if we can statically determine that the loop parameter specification is not a null discrete range. Otherwise, no definitions can be killed by a for-loop. Unlike a while-loop or a for-loop, a repeat-loop (i.e., a loop without an iteration scheme) will always execute at least once. Therefore, a definition killed by the repeat-loop statement block (S_1) is also killed by repeat-loop S. Since the flow control at the end of the loop statement block goes back to the beginning of the loop statement block, the set of definitions that reach the beginning of the loop statement block (i.e., *in*[S_1]) must include the set of definitions generated by the loop statement block (i.e., gen[S1]). Therefore, the set of definitions that reach the beginning of the loop statement block (i.e., *in*[S_1]) is the set of definitions that reach the beginning of the loop (i.e., *in*[*S*]) plus the set of definitions generated by the loop statement block (i.e., *gen*[S_1]).

The variable assignment reaching definitions apply directly to subprograms (FUNCTIONS and PROCEDURES). 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 the 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

[3] The rule for loops stated here is different from the one mentioned in *Compilers*, but it is a consistent derivation with the rule for if-statements.

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 as the posted reaching definitions to be updated later. These sequential signal assignments reaching definitions correspond to and are evaluated in 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:

$$gen[S_2] = post\_gen[S_1] \cup (gen[S_1] - post\_kill[S_1]) \quad (1)$$

$$kill[S_2] = post\_kill[S_2] \cup (kill[S_1] - post\_gen[S_1]) \quad (2)$$

$$out[S_2] = post\_gen[S_1] \cup (in[S_1] - post\_kill[S_1]) \quad (3)$$

For the data-flow equations listed above, S_1 represents the WAIT statement or the end of the PROCESS block while S_2 represents the next statement after the WAIT statement or the PROCESS block where the updated reaching definitions are propagated.

The data-flow equations for updating the *gen*[*S*] and *kill*[*S*] sets are derived from the rule for variable compound statements, shown in Fig. 3(b). Eq. (1) states that the set of definitions generated after the update is the posted set of generated definitions plus the set of generated definitions that are not killed by the posted set of killed definitions. Similarly, Eq. (2) states that the set of definitions killed after the update is the posted set of killed definitions plus the set of killed definitions that are not generated by the posted set of generated definitions. The data-flow equation for updating the *out*[*S*] set is derived from the rule for single variable assignments, shown in Fig. 3(a). Eq. (3) states that the set of definitions that reached the end of this statement after the update is the posted set of generated definitions plus the set of definitions that reached the beginning of the statement that were not killed by the posted set of the generated definitions.

3.3 Concurrent Signal Assignments

Concurrent statements may consist of concurrent signal assignments, such as conditional signal assignment 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 the definitions generated by all the 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*] as 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 Fig. 4(a), Fig. 4(b) and Fig. 4(c), respectively.

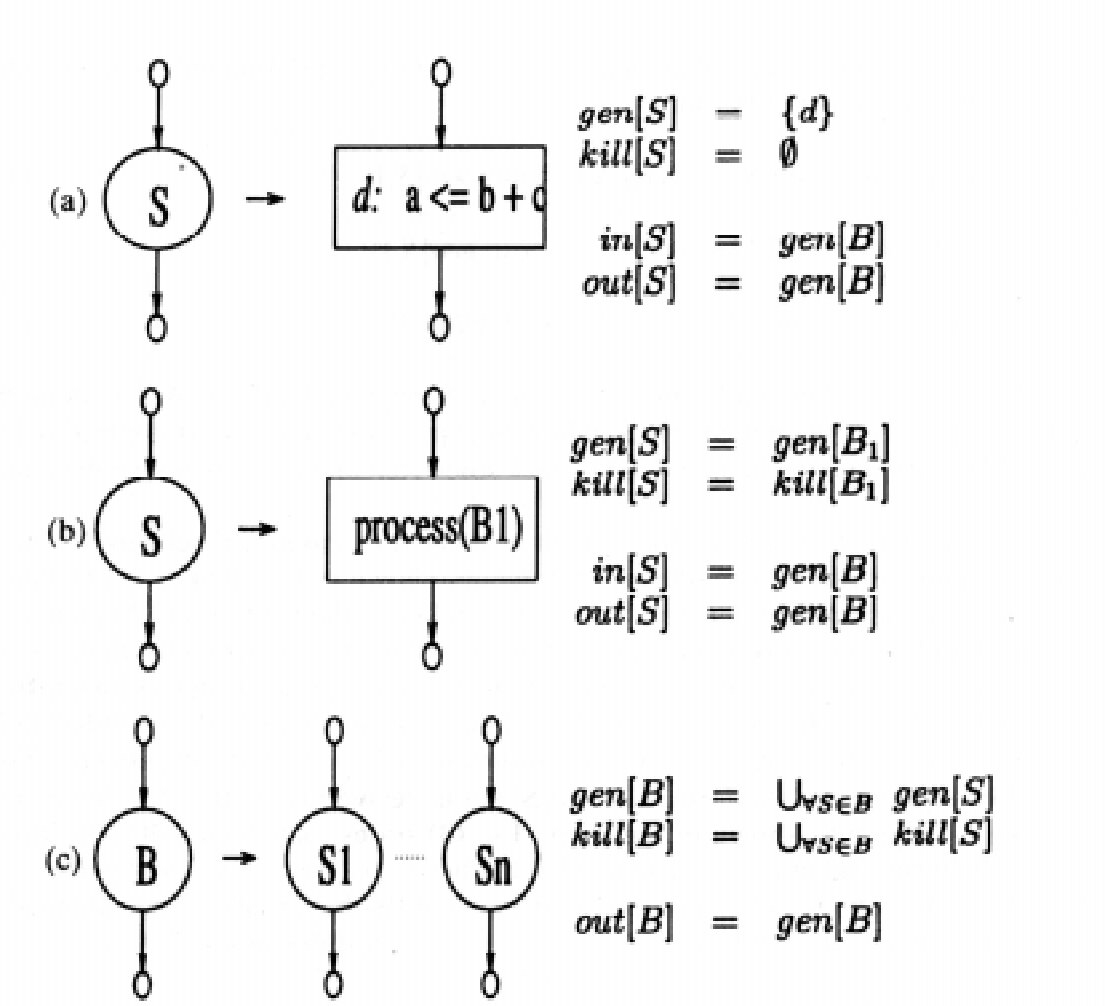

Fig. 4. Data-flow equations for concurrent signal assignment reaching definitions.

Observe the rule for single concurrent signal assignments shown in Fig. 4(a). Like variable assignment, signal assignment ($a <= b + c$) is a definition of signal a, so

the *gen*[*S*] set is this definition *d*. Since the definitions generated by a con-current statement can not be killed by definitions generated by any other concurrent statement, the *kill*[*S*] set is the empty set. For the same reason, the set of definitions generated by the entire concurrent block (i.e., *gen*[*B*]) is also the *in*[*S*] and *out*[*S*] sets.

The rule for process statements, shown in Fig. 4(b), is slightly different from the rule for concurrent signal assignments. Obviously, the *gen*[*S*] set for the process statement is the same as the set of definitions generated by the process block B_1 (i.e., *gen*[B_1]). However, unlike the rule for concurrent signal assignments, the *kill*[*S*] set is not the empty set. The *kill*[*S*] set is the same as the set of definitions generated by the process block B1 (i.e., *kill*[B_1]). The reason is that the definitions for signals in the process block come from the sequential signal assignments which may be killed by any other definition within the process block hierarchy. The *in*[*S*] and *out*[*S*] sets for the process statement are the same as the *in*[*S*] and *out[S*] sets for the concurrent signal assignment.

The rule for the entire concurrent block is shown in Fig. 4(c). The set of definitions generated and killed by the entire block (i.e., *gen*[*B*] and *kill*[*B*]) is the sum of all of the definitions generated and killed by all of the concurrent statements in the block, respectively. The *out*[*S*] set is the set of definitions generated by the concurrent block (i.e., *gen*[*B*]).

3.4 Component Instantiations and Subprogram Calls

The reaching definitions (*gen*[*S*], *kill*[*S*], *in*[*S*], and *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.

4. ANALYSIS ALGORITHMS

Our semantic extraction method consists of several analysis algorithms built upon the control/data-flow analysis technique presented earlier. Each algorithm 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 through 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 the input dependency of a sequential signal assignment and checks if all the 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 the data dependencies and control dependencies. We define the following terms:

data_depend[d_S]: data-flow dependencies for the definition in statement S,
ctrl_depend[*B*]: control-flow dependencies for the statement block B,
input_depend[d_S]: control/data-flow dependencies for the definition in statement S.

The term *data_depend*[d_S] is defined as the assignment expression dependency for the definition in statement *S*. The term *ctrl_depend*[*B*] is defined as the control expression dependency in the entire control path going to the statement block *B*. The term *input_depend*[d_S] is defined as the control and data dependency for the definition in statement S.

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

$$data\_depend_0[d_s] = in[S] \cap \left(\bigcup_{\forall input \in S} D_{input} \right) \quad (4)$$

$$ctrl\_depend_0[B] = in[B] \cap \left(\bigcup_{\forall input \in ctrl\_expr} D_{input} \right) \cup ctrl\_depend_0[P] \quad (5)$$

$$input\_depend_0[d_s] = data\_depend_0[d_s] \cup ctrl\_depend_0[B] \quad (6)$$

$$data\_depend_i[d_s] = \left(\bigcup_{\forall d_j \in data\_depend_{i-1}[d_s]} data\_depend_{i-1}[d_j] \right) \quad (7)$$

$$ctrl\_depend_i[B] = \left(\bigcup_{\forall d_j \in ctrl\_depend_{i-1}[B]} ctrl\_depend_{i-1}[d_j] \right) \quad (8)$$

$$input\_depend_i[B] = \left(\bigcup_{\forall d_j \in input\_depend_{i-1}[d_s]} input\_depend_{i-1}[d_j] \right) \quad (9)$$

Eq. (4), Eq. (5) and Eq. (6) represent the initial iteration for the three dependencies, and they capture the first level of dependency. In Eq. (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 Eq. (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 Eq. (5) captures the control path hierarchy by incorporating the control dependency from the parent block P[4] In Eq. (6), the input dependency for the definition in statement S is the sum of the data dependencies and control dependencies.

Eq. (7), Eq. (8) and Eq. (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). 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 input dependency analysis enables us to perform explicit memory feedback analysis, input signal sensitivity analysis and variable lifetime partitioning. We will discuss these tasks next.

4.2 Feedback Analysis

Feedback analysis determines if a definition has inputs that depend on itself. The term $feedback[d_S]$ is a boolean indicating whether the definition in statement S contains feedback. Eq. (10) summarizes the feedback analysis:

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

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.

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 can 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

[4] 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.

fully specified. For the conditional signal assignment, a simple check for the absence of the ELSE clause without a conditional expression will 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, 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 to 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]) as the set of output signals not assigned in the current sequential block hierarchy B:

pm[*B*]: the 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 the control path from the beginning to the end of the 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 the 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 emory set *pm*[*B*] according to the data-flow equations shown in Fig. 5.

The data-flow equation in Fig. 5(a) states that for each signal assignment in block *B*, the output signal is removed from *pm*[*B*]. The data-flow equations for conditionals are shown in Fig. 5(b). 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 Fig. 5(c). 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. Similarly, for FOR loops, the *pm*[*B*] set is not changed if we can not statically determine whether the loop parameter specification is a null discrete range. However, a REPEAT 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 block B1 of the REPEAT loop.

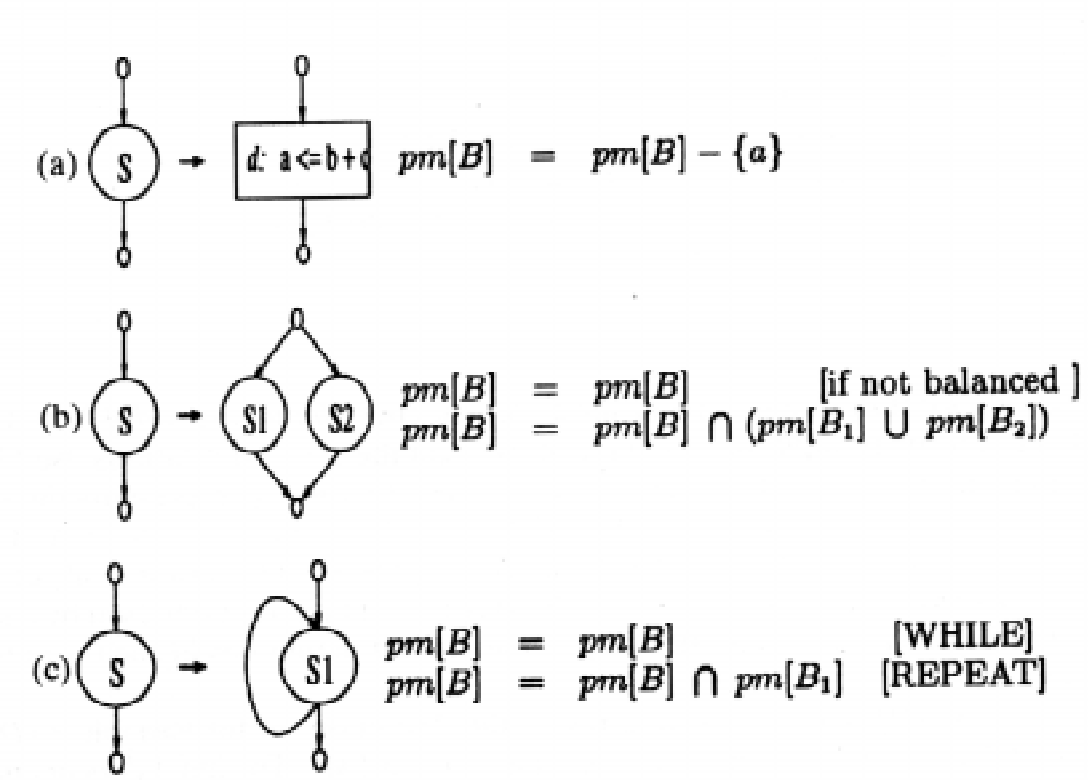

Fig. 5. Data-flow Eqns. for possible memory sets.

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 values 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 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 it is defined as a separate variable in the model abstraction step. The pseudo-code for the partitioning algorithm is listed below:

```
for each depend[d_S] {
    for each input in d_S {
        group_input = D_input ∩ depend[d_S]
        for each d_j in group_input {
            Group[d_j] = Group[d_j] □group_input
        }
    }
}
```

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 in statement *S*. For each variable definition d_j, the term *Group*[d_j] represents the disjoint subset of definitions for the same variable. After 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 the VHDL model shown in Fig. 2, 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*}; therefore, groupd1 is {*d1*}. However, the input dependency for the definition at d3 is {*d2*, *d3*}, therefore, $group_{d2}$ is {*d2*, *d3*}, and groupd3 is {*d2*, *d3*}. As a result, the variable lifetime partitioning for variable *x* is {*d1*} and {*d2*, *d3*}, as shown in Fig. 2 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 the BUS 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 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 done 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 will present the results of semantic extraction for the set of VHDL models shown in Fig. 6. We have kept the models in Fig. 6 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 a comment after each signal or variable assignment.

The results of semantic extraction performed on the models are shown in Table 1. 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.

Table 1. Semantic extraction results.

Experiment	Explict Memory		Implicit Memory		Paritioning	
	Data Dep	Ctrl Dep	Ctrl Path	Sensitivity	Lifetime	Bus Signal
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			

Returning to the semantic extraction experiments, we will present our experimental results starting with the model shown in Fig. 6(a). 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 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 Figs. 6(b) and 6(c) have the same nested control structure. The only difference between the two models is that variables *i* and *j* (Fig. 6(b)) are replaced by signals *x* and *y* (Fig. 6(c)). They demonstrate the subtleties of 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 each process block 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* (Fig. 6(c)) are not assigned in all of the control paths; thus, both signals have implicit memory semantics. Since the process sensitivity list is not specified, all the input signals within the process block hierarchy are implicitly sensitized. Therefore, the input signal sensitivity check is not necessary.

The four models shown in Figs. 6(d), 6(e), 6(f) and 6(g) 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 Fig. 6(d) (and Fig. 6(e)) are similar to the two models shown in Fig. 6(g) (and Fig. 6(f)), with variables *i*, *j* and *k* replaced by signals *x*, *y*, and *z*. The two models shown in Fig. 6(d) (and Fig. 6(g)) have an initial variable assignment (and signal assignment) at d5.

For the model shown in Fig. 6(d), 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 definition d4. Consequently, the definition at d4 has explicit feedback due to the control dependencies on input variable *k* from the if-expression.

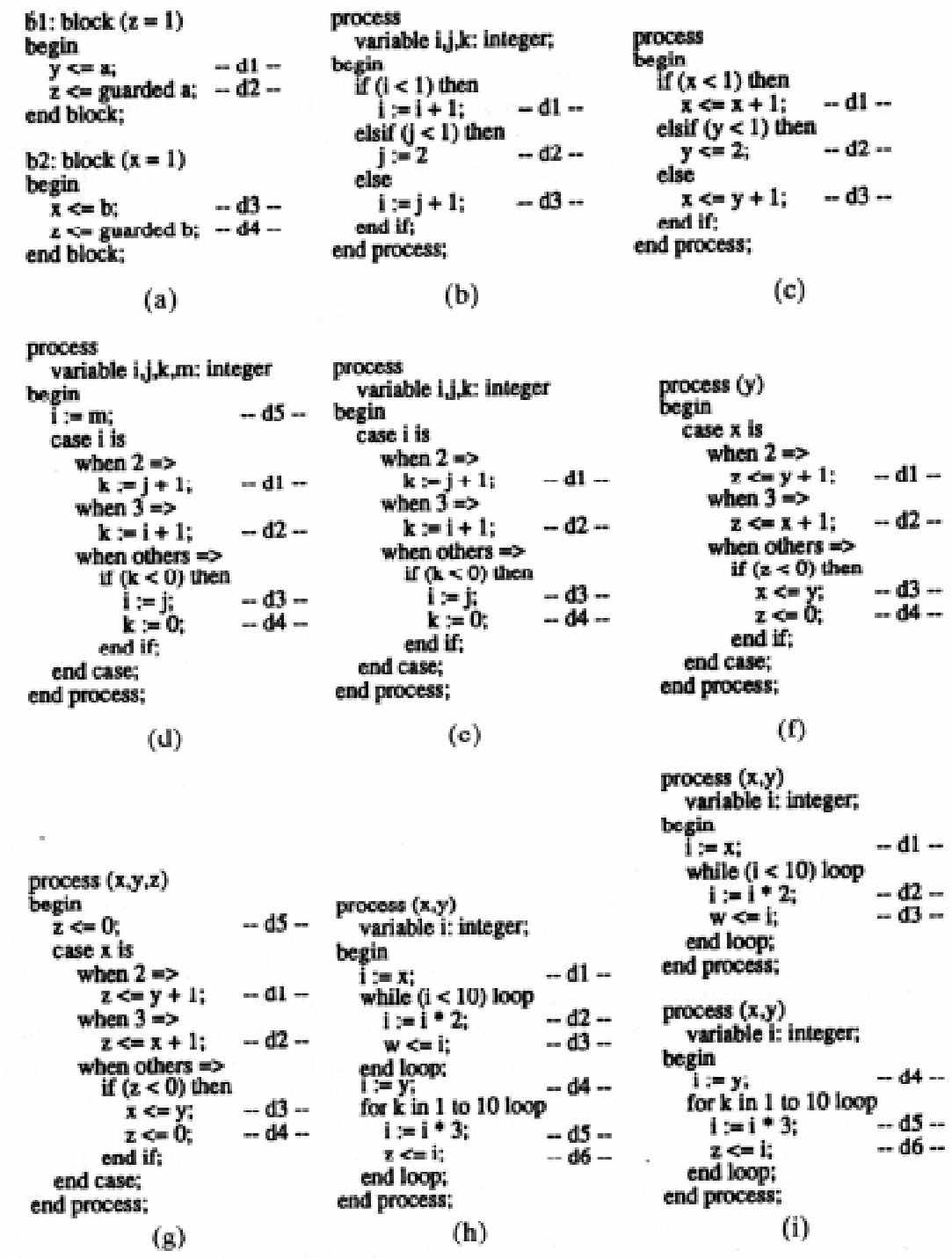

Fig. 6. Semantic extraction experiments.

For the model shown in Fig. 6(e), removal of the variable assignment at d5 causes 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 Fig. 6(f) has data dependency and control dependency similar to that presented in the previous model. A balanced conditional signal assignment check performed on the model reveals 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 reveals 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 Fig. 6(g), 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 reveals that only signal *x* has implicit memory semantics due to unbalanced conditional signal assignments. An input signal sensitivity check reveals that all the 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 Fig. 6(h) 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 Fig. 6(i). 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. Analyzing the abstract state space of the model and applying our model abstraction techniques [6], the result is a model with abstractions that reduces the number of states necessary to perform formal verification. Using our semantics based model abstraction method, we have obtained [6] a 10^2 to 10^{11} fold reduction in state space size and at least an 18 fold reduction in verification time compared to those of unreduced models. More importantly, the abstract models allow us to verify some models that are 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

1. Z. Har' El and R. P. Kurshan, *COSPAN User's Guide*, AT&T Bell Laboratories, 1993.
2. K.L. McMillan, *Symbolic Model Checking*, Kluwer Academia Publisher, 1993.
3. R.K. Brayton, A. Sangiovanni-Vincentelli, F. Somenzi and A. Azi S. Sarwary, "Vis: A system for verification and synthesis," in *Proceedings of Conference on Formal Methods in Computer-Aided Design*, 1996, pp. 428-432.
4. IEEE, New York, *VHDL Language Reference Manual*, IEEE Standard 1076-1993, 1994.
5. A.V. Aho, R. Sethi and J.D. Ullman, *Compilers: Principles, Techniques and Tools*, ch. Code Optimization, Addison-Wesley Publishing Company, 1986, pp. 608-623.

6. Y.-W Hsieh and S.P. Levitan, “Model abstraction for formal verification,” in *Proceedings of The 1st Design, Automation and Test in Europe Conference*, 1998, pp. 140-147.

Yee-Wing Hsieh□□□□□is Ph.D. student at the University of Pittsburgh. He received his B.S. degree in Electrical Engineering from Ohio State University (1989) and his M.S. degree in Electrical Engineering from the University of Pittsburgh in (1992). He has worked for both Mentor Graphics and AT&T Bell Laboratories as a summer intern. His research interests include formal verification, high-level synthesis, HDL simulation and synthesis for VLSI. He is a member of ACM and IEEE.

Steven P. Levitan is the Wellington C. Carl Associate Professor of Electrical Engineering at the University of Pittsburgh. He received his B.S. degree from Case Western Reserve University (1972) and his M.S. (1979) and Ph.D. (1984), both in Computer Science, from the University of Massachusetts, Amherst. He worked for Xylogic Systems designing hardware for computerized text processing systems and for Digital Equipment Corporation on the Silicon Synthesis project. He was an Assistant Professor from 1984 to 1986 in the Electrical and Computer Engineering Department at the University of Massachusetts. In 1987 he joined the Electrical Engineering faculty at the University of Pittsburgh. His research interests include VLSI architectures, optoelectronic computing systems, parallel algorithm design, and HDL simulation and synthesis for VLSI. He is an Associate Editor of the ACM Transactions on Design Automation of Electronic Systems. He is Chair of the ACM Special Interest Group on Design Automation, a member of OSA, and a senior member of IEEE/CS.