PBS User’s Guide

Version 2.2
F. Brewer, A. Crews, A. Seawright
1.0 Preface

PBS stands for production-based specification. This guide and reference describes the PBS format and a tool that compiles designs specified in this format.

This tool was developed at the University of California, Santa Barbara Department of Electrical and Computer Engineering. The project that led to its development was made possible through the generous support of Synopsys Inc., the California Micro Program, and the National Science Foundation. This documentation is Copyright © 1997 by The University of California. It may be freely distributed as long as it is distributed in complete form with this copyright notice. Originating work was performed by Andrew Seawright, optimization algorithms and extensions were made by Andrew Crews, both under the direction of Forrest Brewer.

2.0 Introduction to PBS

2.1 Overview

The use of high-level and hardware description languages is now widespread in the specification, verification, and synthesis of digital hardware. These languages allow the abstraction of circuit-level details in the design process and enable design specification at the “algorithm-level”. High-level language specification provides numerous advantages including increased design productivity, technology independence, and greater reuse of design specifications.

PBS addresses the specification, design, and optimization of certain types of complex control-dominated designs. For the most part, conventional hardware description languages in use today model behavior in a procedural fashion similar to a procedural programming language. In a procedural style, the control flow of the design is specified explicitly. Description of control flows typically use branching constructs such as if-then-else and case statements, which may be functions of explicitly specified control state variables. Design systems using this kind of specification have become commercially available and are successful tools. However, for a class of problems, the specification of the machine
behavior in this format is unnecessarily lengthy and complex. This is especially true for
problems in which the time sequence behavior is complex, in that the control state space is
large or difficult to describe explicitly (e.g. protocol decoders, communication devices,
computer interface subsystems).

An alternative to a procedural specification style is an applicative or declarative style in
which the legal behaviors of the design are described through a hierarchical composition
of the legal sub-behaviors. The salient characteristic of this Production-Based Specifica-
tion (PBS) is its implicit representation of the control flow of the design, in an applicative
design style.

This applicative style of specification is intended to avoid some of the problems associated
with explicit description of the control flow. An applicative style, for instance, does not
necessarily require encoding of control states and state transition in the high level descrip-
tion, as may be required in procedural design style. Additionally, procedural specifications
of complex, control-dominated design can be error-prone since, in general, the resulting
descriptions are correspondingly complex, while applicative style descriptions often do
not suffer from this problem. Finally, incremental modifications of a design described in
an applicative style are much less likely to require large global changes of the design, as
may occur in procedural styles.

The essential idea behind PBS is to represent the control structure of the design using a
hierarchical set of productions. These productions describe the desired “protocol” at the
high-level, and “recognition” of these productions effect appropriate actions.

Designs specified in PBS are really design entities. These design entities are assumed to
interact with other design entities, which may or may not be specified using PBS, as com-
ponents of a larger system. PBS format is compatible with the VHDL [1] design method-
ology and is intended to be used by the designer as an alternative specification style used
in conjunction with other specification and design methods.

The compiler also supports other options for encoding the controller mechanism in a non-
code generation mode. In this mode, the resulting controller is written out as a BLIF or
ATT format file with latches and an appropriate reset state. In this mode, actions have a
symbolic meaning and are currently not-hot encoded. Extensions to allow output encoding
in the language and moore-based persistent outputs will be available soon.

2.2 PBS Format

The PBS specification is a single file comprised of three parts. The first part is called the
“environment and support” section. This section contains information about the interface
of the design, supporting VHDL code, construction and compilation directives, and defini-
tions and declarations to support action clauses in the productions. The second part of the
PBS specification contains the declaration of the input signals, productions, markers, and
declared actions used in the production section. These declarations simplify the task of correct compilation analysis and eliminate the problems of predefined key-words. The last part of the specification consists of a collection of productions describing the behavior of the design entity. Since this third part is the most important part of the specification, we focus on it first.

2.3 Production Definition Section

2.3.1 Overview - Productions, Operators, and Actions

A production is a named composition of symbols, operators, and action clauses. There are two kind of productions, those for building up sequential behaviors, and those constructing combinational Boolean functions. The symbols in a sequential production are either tokens or references to other sequential productions. A token is a reference to a Boolean production in a sequential production. The symbols in a Boolean production are either references to other Boolean productions or atomic symbols. Atomic symbols are Boolean functions of input interface signals and marker evaluations or are language defined symbols.

Several different types of composition operators are used to compose the productions, and they are used to build more abstract or complex productions from simpler productions. For example, they create productions of more complex behaviors from productions of simpler ones. The composition operators are similarly grouped into sequential and combinational types for use in the two kinds of productions.
A token is “accepted” or “recognized,” if its Boolean function is satisfied in the cycle the token is referenced through execution of the productions. A production is accepted when the time sequence of behaviors dictated by its composition is satisfied. Symbols, compositions, and productions are annotated with action clauses, or actions for short. Basically, an action is a specified data-flow behavior which is executed when its antecedent (symbol, composition, or production) is recognized.

### 2.4 Introductory Example

A set of productions are shown below. There are five productions in this description:

```
mealy {}
::
#input xc xd
#production mouse event forward reverse rising
::
mouse -> .*, event;
event -> forward || reverse;
forward -> (xd):rising; [ x <= x + 1; ]
reverse -> (~xd):rising; [ x <= x - 1; ]
rising -> ~xc+, xc;
::
```

![Figure 1.](image)

**Figure 1.**

mouse, event, forward, reverse and rising. All are sequential productions with forward and reverse events initialized simultaneously. The symbols xc and xd are input signals from the interface to the external world. The first production in a set of productions, in this case the mouse production, is called the *top-level production*. The top-level production encompasses the behavior of the whole design. Other than the top level production, all production may occur in any order.

This set of productions implement a mouse position decoder machine, which continuously updates the variable x with a current 1-dimensional position based on the quadrature encoding of the signals xc and xd from external motion sensors. The position is updated if the productions forward or reverse are recognized. Due to the structure in this
example, these productions could be recognized, starting from any present state of the machine. In general, several productions may be recognized or be in the process of recognition simultaneously. The bottom portion of the figure shows an example of the time sequence behavior of this design.

PBS specifications describe synchronous designs where behavioral changes are synchronized to transitions on global clock signals. In the timing diagram in Figure 1, the clock is not shown, however at least one clock is always assumed. Synchronous token recognition provides the atomic mechanism for the time sequence behavior of a design.

The syntax of a production is as follows:

<production identifier> -> <production composition>; [action]

This syntax is based on the Backus-Naur Form (BNF) [3] used to describe language grammars. The production identifier is a unique name labeling the production. This name can be used in other productions to refer to it. The production composition contains the symbols, composition operators, and actions defining the meaning of the production. The production composition is terminated with a semicolon, however, it is possible to place an action list after the semicolon, in which case the actions are referenced to the production identifier.

The symbols in the production composition may reference other productions. When a production composition references another production, by its identifier name, it is just as if the referenced production’s composition was substituted in place of the identifier name. Production referencing has several advantages: it allows decomposition of design behavior into logical portions, it allows for more efficient descriptions through sharing and reuse of productions, and it provides more flexibility in the placement of actions.

When productions reference other productions among the collection of productions, this referencing may not be recursive. In a recursive set of productions, the production referencing contains at least one cycle. This means that the hierarchical composition of the design cannot be described using a directed-acyclic graph (DAG). Productions are not allowed to be recursive because one can’t guarantee, in general, that the specification is of a finite machine1 (machines not requiring an arbitrary amount of memory storage [3]). Recursion in software specification is essential for the description of language grammars, which are typically push-down or LALR automata. However, PBS is oriented specifically for the hardware domain where we are interested in describing the behavior of finite machines (FSM’s, controllers, protocols), and recursion is not essential, nor necessarily efficient. One type of behavior commonly modeled by recursion is the use of non-local referencing in which the recognition of a production might depend on the coincident rec-

---

1. Although tail recursion does not jeopardize the finite machine requirement, it is still not allowed. This is not a problem, since tail recursive behavior can be concisely described using the Kleene Closure operator "*".
ognition of an unrelated production. In PBS, this behavior is achieved using a marker, which is a special type of internal action that creates a Boolean signal. Since this signal can be used to enable production recognition, this behavior is easy to achieve.

If a PBS specification contains recursion, an error will be generated at compile time indicating the line number of the first occurrence of the production recursed. For example, the following set of productions is recursive:

```plaintext
a -> b || c, b;
b -> ~x & z;
c -> a, b;
```

If compiled, the error would indicate recursion into the production on the first line.

In the mouse example, the sequential operators: “,”, “*”, “+”, and “|”, are extensions of the classical Regular Expression operators, and the operators: “;” and “~” are Boolean operators, (“;” allows a Boolean token to qualify a production). These operators and others are tabulated on the next page in Table 1. Each of the operators will be described in detail later.

As mentioned earlier, there are two types of productions: Boolean and sequential productions. There are Boolean and sequential composition operators, as well. With the assumption of non-recursive productions, the following statements apply.

- A Boolean composition contains only Boolean composition operators and references to Boolean productions.
- A sequential composition must contain at least one sequential composition operator or at least one reference to a sequential production.
- Sequential productions can be referenced from sequential compositions.
- Boolean productions can be referenced from Boolean compositions.
- Boolean productions can be referenced from sequential compositions, and when done this reference is a reference to a token.
- Sequential productions cannot be referenced from Boolean compositions.
2.5 Composition Operators and Examples

In this section, the various composition operators will be described. These composition operators are tabulated in Table 1 above. In this table, \(<\text{comp}>\) refers to a sequential composition or a sequential production reference or a token reference. \(<\text{bool-comp}>\) refers to a Boolean composition or a Boolean production reference. Finally, \(<\text{int}>\) refers to an integer argument.

<table>
<thead>
<tr>
<th>operator</th>
<th>name</th>
<th>type</th>
<th>usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>,</td>
<td>concatenation</td>
<td>sequential</td>
<td>(&lt;\text{comp}&gt; , &lt;\text{comp}&gt;)</td>
</tr>
<tr>
<td>^</td>
<td>multiple concatenation</td>
<td>sequential</td>
<td>(&lt;\text{comp}&gt; ^ &lt;\text{int}&gt;)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>sequential or</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>sequential and</td>
<td>sequential</td>
<td>(&lt;\text{comp}&gt; &amp;&amp; &lt;\text{comp}&gt;)</td>
</tr>
<tr>
<td>!</td>
<td>sequential not</td>
<td>sequential</td>
<td>(! &lt;\text{comp}&gt;)</td>
</tr>
<tr>
<td>*</td>
<td>Kleene Closure</td>
<td>sequential</td>
<td>(&lt;\text{comp}&gt; *)</td>
</tr>
<tr>
<td>+</td>
<td>one-or-more</td>
<td>sequential</td>
<td>(&lt;\text{comp}&gt; +)</td>
</tr>
<tr>
<td>! !</td>
<td>exception-handler</td>
<td>sequential</td>
<td>(&lt;\text{comp}&gt; ! ! &lt;\text{comp}&gt;)</td>
</tr>
<tr>
<td>!R</td>
<td>exception-reset</td>
<td>sequential</td>
<td>(&lt;\text{comp}&gt; !R)</td>
</tr>
<tr>
<td>!E</td>
<td>exception-excitation</td>
<td>sequential</td>
<td>(&lt;\text{comp}&gt; !E)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Boolean or</td>
<td>Boolean</td>
</tr>
<tr>
<td>&amp;</td>
<td>Boolean and</td>
<td>Boolean</td>
<td>(&lt;\text{bool-comp}&gt; &amp; &lt;\text{bool-comp}&gt;)</td>
</tr>
<tr>
<td>^</td>
<td>Boolean exclusive-or</td>
<td>Boolean</td>
<td>(&lt;\text{bool-comp}&gt; ^ &lt;\text{bool-comp}&gt;)</td>
</tr>
<tr>
<td>~</td>
<td>Boolean complement</td>
<td>Boolean</td>
<td>(&lt;\text{bool-comp}&gt; ~ &lt;\text{bool-comp}&gt;)</td>
</tr>
<tr>
<td>:</td>
<td>qualification</td>
<td>special</td>
<td>(&lt;\text{bool-comp}&gt; : &lt;\text{comp}&gt;)</td>
</tr>
<tr>
<td>x (2)</td>
<td>delay</td>
<td>special</td>
<td>(&lt;\text{marker-identifier}&gt; ( &lt;\text{int}&gt; ))</td>
</tr>
</tbody>
</table>

2.5.1 Sequential Operators

When a composition involves a sequential operator, the operator is recognized (accepted) if its composition operands satisfy some requirements based on the type of operator.

2.5.1.1 Concatenation “,”

The concatenation operator concatenates the behavior of its two composition operands together in time sequence. It is helpful to view these composition operands as sub-machines. The acceptance or recognition of the concatenation of two sub-machines occurs if the first sub-machine is started and is recognized immediately, followed by the initiation and acceptance of the second sub-machine. For example, in the production:
Using the PBS Synthesis System

p -> a, b;

p is recognized if the a sub-machine is recognized, followed by the initiation and recognition of b. Here, a and b are defined by other productions and compositions. The concatenation operator is used to build behaviors responding to activities occurring in order. Three additional examples are:

```
frame     -> start_bit, data_bits, stop_bit;
three_blocks -> start_block, middle_block, end_block;
instruction -> op_code, operands;
```

2.5.1.2 Multiple Concatenation “^”

Multiple concatenations of the same composition a fixed number of times can be specified using the multiple concatenation operator. This operator concatenates the left hand side composition operand an integral number of time specified by the right hand argument. For example in the production:

```
byte -> bit^8;
```

a byte is the recognition of eight bit’s in a row. This operator is a convenient shorthand useful for cases where using the previous concatenation operator would be tiresome. For example, the following production is the long way to write the same behavior as the production above.

```
byte -> bit, bit, bit, bit, bit, bit, bit, bit;
```

Since PBS specifies a chain of flip-flops for these kind of constructions, rather than using encoded counters, for example, it can be very useful to perform factoring on multiple concatenation constructions. Placing parenthesis around the integer indicates that the production should be considered for sequence factoring by the compiler. The implementation of sequence factoring is not defined, only that the behavior is equivalent.

```
byte -> bit^(8);
```

2.5.1.3 Sequential Or Operator “| |”

The sequential or operator is used to specify a choice of desired behaviors. The recognition of the left hand composition operand or the right-hand composition operand or both will cause the acceptance of the two-way or composition. For example:

```
word -> control_word || data_word;
```
If active, recognition of the production word will occur if control_word is recognized or data_word is recognized. Some more example productions using “||” are:

\[
\begin{align*}
\text{event} \rightarrow \text{left} \mid \mid \text{right}; & \quad // \text{from the mouse example} \\
\text{block} \rightarrow \text{a_block} \mid \mid \text{b_block} \mid \mid \text{c_block}; & \quad // \text{three-way or} \\
\text{message} \rightarrow \text{x}^{12} \mid \mid \text{y}^{8}; & \quad // \text{or used with other operators} \\
\text{foo} \rightarrow \text{(bar, baz)} \mid \mid \text{q}; & \quad // \text{or used with other operators}
\end{align*}
\]

The sequential or operator, as well as the sequential and operator, described next, build concurrence into the design.

2.5.1.4 Sequential And Operator “&&”

A composition containing the sequential and operator is recognized when its left hand composition operand and its right hand composition operand are recognized at the same time (in the same cycle). This operator can be used for synchronization. For example, in the following production:

\[
\text{p} \rightarrow \text{event1} \&\& \text{event2};
\]

\(p\) is recognized only if event1 and event2 are recognized together in the same cycle. To expand on this example, replace the event1 and event2 references with the compositions \((\neg x^+, x^+)\) and \((\neg y^+, y^+)\):

\[
\text{p} \rightarrow (\neg x^+, x^+) \&\& (\neg y^+, y^+);
\]

where \(x\) and \(y\) are interface signals. The composition \((\neg x^+, x^+)\) is recognized if a rising edge is “seen” on the signal \(x\) (see descriptions of the operators “+” and “~”) Similarly, \((\neg y^+, y^+)\) recognizes a rising edge on the signal \(y\). The compositions replacing event1 and event2 were “built” to continuously accept after the rising edge on their respective signals, so long as the signals remain high. Thus, the production \(p\) is recognized only if a rising edge occurs on both \(x\) and \(y\). These rising edges may arrive in either order or in the same cycle. One final note, the production \(p\) will continue to be recognized as long as the signals \(x\) and \(y\) remain high.

2.5.1.5 Kleene Closure Operator “*”

The Kleene Closure operator is an operator for the specification of variable sequences. Specifically, zero or more concatenations of the operand composition will be recognized. For example in the production:

\[
\text{p} \rightarrow \text{a*}, \text{b};
\]
zero or more a’s followed by b will cause the recognition of p. For example, the following sequences will all be recognized: { b, ab, aab, aaab, ... }. Here, a and b could be tokens or behaviors based on other productions.

A common use of the Kleene Closure operator is with the special symbol “.” (described later). For example, in the production:

```
    p -> .*, x;  // any sequence ending with x.
```

The behavioral idiom “. *, x” means “anything followed by x” or “anything ending with x”, where x could be a token, a composition, or another production. Note that, “anything ending in x” describes sequences that may contain embedded x’s, e.g. “abxbaxxx”. This behavioral idiom is used in the top-level production of the mouse example. The mouse recognizes “any previous encoder sequence followed by an event”. In general, compositions and productions using the “*” operator and the “+” operator (described below) have the property that they recognize repeatedly, or describe periods of continuous recognition.

### 2.5.1.6 One or More Operator “+”

The “+” operator is similar to the Kleene Closure operator, however, compositions involving it recognize one or more concatenations of the operand composition. The composition “x+” is equivalent to “x, x*” and “x*, x”. However, it is far more efficient to use “+” than the concatenation since “+” creates one control point in the worst case. For example:

```
    p -> ~x+, x;  // p recognizes a rising edge on the
    // interface signal x
```

```
    p -> ~x+, x+;  // p recognizes a rising edge on x,
    // and continuously recognizes if
    // x remains high
```

### 2.5.1.7 Sequential Not “!”

The sequential not operator is used to invert the state of recognition of its single composition operand. The operator is in a state of recognition only if its composition operand is not in a state of recognition. For example, recognition will occur if: the operand composition has not yet been initiated, it was initiated but is not yet accepting, or was initiated and will never accept. This operator is useful in certain special cases, such as in, the following production:

```
    p -> !(.*, a, a, .*);  // accept any sequence without
    // two a’s in a row
```

This production recognizes any sequence not containing two a’s in a row. The a’s could be tokens or behaviors described by other productions.
2.5.2 Overview - Exception Operators - “!!”, “!R”, and “!E”

Exception operators are useful for specifying exception handling, recovery, and re-synchronization mechanisms as well as closing incomplete specifications. These operators are used to specify behaviors based on the conditions in which other behaviors, or sub-machines, enter states from which they can never accept. In a sense, behaviors can be a function of what is not described by a composition or production. This allows access to the complement space of compositions or productions. These operators are similar to the sequential not operator, described above, however, the exception operators describe behavior solely in terms of the conditions in which other behaviors enter states from which they can never accept. Precisely, if the subject composition describes a set of acceptable event sequences, an exception will accept on any event sequence which will leave the set on the next cycle.

2.5.2.1 Exception Operator with Handler “!!”

If a composition of this operator is initiated, execution begins with the left-hand composition sub-machine. If this sub-machine is about to enter a state from which it can never accept, in the next cycle, this operator will cause the initiation of another sub-machine, called the handler, specified by the right-hand composition operand. If the handler sub-machine accepts, the machine behaves exactly as if the left hand machine accepted, and the complete composition is recognized. For example, in the production:

```
data -> data_block!!handler_machine;
```

the data production will be recognized if data_block is recognized, or if the data_block sub-machine enters a state from which it can never accept, followed by the acceptance of the initiated handler Machine sub-machine. The “!!” operator, as well as the other exception operators, can be nested. For example, the composition of the handler_machine may contain imbedded exception handlers.

2.5.2.2 Exception Reset Operator “!R”

Initiation of a composition containing the “!R” operator begins with the execution of the modified operand sub-machine. The exception reset operator, will re-initiate (reset) the sub-machine if, in the next cycle, it will enter a state from which it can never accept. In other words, the operand sub-machine is constructed normally except that instead of entering an end state from which it can never except, it resets, and behaves as if it had not yet been initiated. If the operand composition sub-machine does accept, the complete composition accepts. This operator is useful for behaviors that “abort back to the beginning until they are satisfied.” For example, in the following production:

```
message -> preamble!R, body, tail;
```
when the message sub-machine is initiated, recognition of a preamble will be attempted. If the preamble can’t be recognized for some reason, the preamble sub-machine will be restarted. Once the composition “preamble!R” recognizes the first valid preamble present in the input data stream, execution will continue with the initiation of the body sub-machine.

2.5.2.3 Exception Excitation Operator “!E”

A composition with “!E” is accepted only when the sub-machine built from its composition operand will go into a state from which the sub-machine will never accept in the next cycle. This operator is similar to the sequential not operator except that the sequential not operator does not distinguish between the cases of “not currently accepting” and “will never accept”.

2.5.3 Boolean Operators “&”, “|”, “~”, and “^”

The Boolean operators: Boolean and “&”, Boolean or “|”, Boolean not “~”, and Boolean exclusive-or “^”, are used in Boolean compositions for specification of Boolean functions. Boolean functions are used as tokens in sequential productions and used as the left-hand operand in the qualification operator described in the next section. Note, the operator “^” has two meanings, i.e. it is an overloaded operator. The symbol “^” is also used for the sequential multiple concatenation operator. The meaning is never ambiguous since the right-hand operand distinguishes between the two cases, e.g. if it is an integer it must be the sequential concatenation operator. The following production is a Boolean production from the mouse example:

```
A -> xc & ~xd;
```

2.5.4 Qualification Operator “:”

A sequential production or composition may be qualified with a Boolean production or composition using the special qualification operator. The qualification operator modifies the sequential behavior of the right-hand sub-machine operand with the Boolean function represented by the left-hand Boolean composition operand. The behavior of the qualified sub-machine is the same as that of the unqualified sub-machine, except that, in order for the qualified sub-machine to be recognized, the Boolean function qualifying the sub-machine must remain true throughout the sub-machine’s execution. In other words, the behavior of the qualified sub-machine is the same as the unqualified sub-machine in which all of its token functions have been anded with the qualifying function.

The qualification operator is useful because it can modify or refine the behavior of a production for different contexts. For example, a “generic” sub-machine can be specified. The sub-machine’s production identifier can be referenced from several other productions in different contexts, and its behavior can be refined through qualification. This allows con-
Using the PBS Synthesis System

These productions describe a portion of a serial asynchronous receiver protocol, defining the makeup of a valid data frame. In this protocol, the number data bits in the frame may be seven or eight depending on the receiver’s mode. Here, qualification is used to modify the generic bit sub-machine for the two modes the receiver could be in: mode1 or mode2. The truth of the Boolean productions mode1 and mode2 reflect the current mode. They are functions of the signals x0, x1, and x2, from, for example, an external control register. In order for a valid frame to be recognized, seven valid data bits must be received with the condition that mode1 is true throughout, or eight valid data bits must be received with the condition that mode2 is true throughout. The makeup of a bit is determined by the bit production elsewhere in the specification.

2.5.5 Delay (Marker) Operator

The delay operator facilitates expansion of the context of a production by allowing implicit reference to current or past values of markers. A delay is a Boolean operator (returns a Boolean value) whose value is a sample of the acceptance state of the production to which the marker is attached. The sample is referenced from the execution of the current production with a given delay. For example, foo(1) is a Boolean variable which is true if and only if <foo> marked a production accepted in the previous cycle. Similarly, foo(3) is true iff the production <foo> marked, accepted three cycles ago. The value of a delayed production is defined to be zero if the production has not been in current context. i.e. 1 means accepted, zero means either did not accept or has not been in current context.

Delays can have a reference time of zero, indicating that the sample is to be taken on the conclusion of the current cycle. In effect, the named productions are retimed to the previous cycle and the resulting mealy edge is made available for use in the current production composition. Clearly, there is a danger here of creating combinational loops and/or inconsistent logic. However, the power of being able to directly access the current state is a great convenience. Unfortunately, determining precisely when such an assignment might lead to inconsistent machine behavior is quite difficult. In current PBS, a check is made which warns the user if the compiler suspects that a marker might be used inconsistently.

Delay operators have great utility in certain machine constructions and create a simple method for implementing conventional state tables as productions. This use does not alter
the theoretical power of the language -- but can be very expressive in some contexts. Please note however, that sequential complement operations may produce valid but unexpected behavior if delay operations are used on interior sub-machines. In particular, the reset exception can restart a submachine -- but won’t clear the delayed values of the productions.

2.5.6 Operator Precedence

Table 2, tabulates the precedences and associations of the various operators. Beginning with the “||” and “!!” operators, this table runs from lowest to highest precedence, where operators of equal precedence appear on the same line.

<table>
<thead>
<tr>
<th>operators</th>
<th>association</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>left</td>
</tr>
<tr>
<td>!</td>
<td>right</td>
</tr>
<tr>
<td>,</td>
<td>left</td>
</tr>
<tr>
<td>* + !R !E</td>
<td>left</td>
</tr>
<tr>
<td>:</td>
<td>left</td>
</tr>
<tr>
<td>^ (both)</td>
<td>left</td>
</tr>
<tr>
<td>&amp;</td>
<td>left</td>
</tr>
<tr>
<td>~</td>
<td>right</td>
</tr>
<tr>
<td>foo()</td>
<td>left</td>
</tr>
</tbody>
</table>

These precedences may be overridden by appropriate grouping of terms using parenthesis, as in this example:

```
mode2 -> ~x0 & (x1 || x2);
```

2.6 Actions

In PBS there is a partitioning of the design’s behavior between the control-flow, or “protocol” of the design, and the data-flow computations that are performed in response to the flow-of-control. The control-flow in a PBS specified design is described *implicitly* by the structure and composition of the production sub-machines. Mainly, it is this implicit description of the control-flow that makes this style of specification an attractive alternative to the designer. In order for the designer to “hook-in” to the implicitly described control-flow when specifying the data-flow, both a method and methodology are needed. The action clauses and the rules that govern their use serve this purpose.
Action clauses annotate PBS compositions and productions. An action is a segment of the target code (usually VHDL) that describes some data flow behavior to be executed when its antecedent is recognized. In PBS, the antecedent is comprised of conditions that govern acceptance based on token, composition, and production recognition.

2.6.1 Types of Actions

There are two kinds of actions: actions whose execution ultimately occurs inside a combinational VHDL process, and actions intended for sequential VHDL processes. The structure of the output VHDL is described in a later section. The type of each action clause is explicitly stated by the designer. This allows the designer more flexibility and control over how the VHDL constructs are interpreted and implemented, but requires a certain level of understanding of how the PBS specification is compiled and optimized.

A combinational VHDL process is one that describes behaviors that can be implemented as a combinational logic circuit. A sequential VHDL process is one that depends on a global clock and is likely to describe behaviors that require flip-flops or registers to save operands (data-flow) state from cycle to cycle. These memory elements are not explicitly stated, but rather are inferred from the design when synthesized. For more information on how VHDL processes are synthesized see [2].

Actions for a combinational process are enclosed in curly braces, “{...}”. This kind of action assumes that the results of its computations are only needed for the duration of the cycle in which the action is executed. These actions do not require the storage of data-flow state for use in another cycle, e.g. the state of action variables and signals need not be saved. Actions of this type, for example, assert combinational control signals.

Actions for a sequential process are enclosed in square brackets, “[...]”. These actions are used when action execution requires that results must be saved because they are needed in another cycle. Likely, these results are used by other actions in other portions of the protocol. Thus, square bracket actions are used for actions that must save data-flow state. For example, in the mouse example, the actions are of this type because the value of the position variable \( x \) must be saved and reused in subsequent cycles.

Any named action causes a symbolic output to be built in the BLIF or ATT format controller output. Currently, each such output is one-hot encoded and each named action creates a BLIF output signal. There is no distinction between combinational and sequential actions in the output format.

2.6.2 What Can Go in an Action?

In general, any legal VHDL code that can appear inside a VHDL process may appear in a PBS action. The way in which the output VHDL generated by the PBS compiler is further processed, however, will influence the VHDL constructs used in the actions. Usually,
VHDL synthesis will impose tighter constraints than simulation on the action clauses. For information on these issues, see [2]. As described above, interpretation of the action depends on the kind of VHDL process which is generated.

### 2.6.3 Where Can Actions Go?

Action clauses may be placed: immediately following a token, immediately following a reference to a sequential production, immediately following a parenthesized sequential composition, or attached to an entire production (after the semicolon). The behavior of the placed action is such that it is executed in the cycle that the respective annotated sub-machine accepts. For the four placement cases, the respective sub-machines are: the token sub-machine, the sub-machine represented by the reference to the production, the sub-machine represented by the parenthesized composition, or the sub-machine represented by the production. Actions should not be placed in Boolean compositions or attached to Boolean productions. The following are example action placements:

```vhdl
mmm -> (~x0 & x1) {z <= 0;}  
foo {z <= 0;}

foo -> x0 & x1 & x2 [z := z + 1];
|| bar
  [w <= z;] [x <= ‘1’;] { y <= ‘0’;}

bar -> (event1 && event2) { z <= w;}
```

### 2.6.4 Behavior and Action Execution Model

The behavior of the PBS design entity, when viewed as a “black-box” component, is defined by the execution of the implicit production control-flow, such that the actions are executed at their respective points in the protocol, and the execution of each action occurs entirely within the accepting cycle. Thus, the behavior model assumes that the black-box behavior is equivalent to the design implementation in which each action is implemented as a combinational logic function. For sequential process actions, only the outputs of the combinational functions implementing the action may feed into registers or latches to save data-flow results.

The previous paragraph described how the behavior of the design is defined, not how the design is implemented. Compilation, synthesis, and optimization may transform the design, but these transformations will not change the “black-box” behavior defined above. The behavior model defines behavior, based on the fixed combinational execution of the actions. PBS synthesis and register transfer level (RTL) and logic optimization may: break actions apart into smaller operations, spread actions across several cycles, share resources between actions, and re-schedule, re-time, or pipeline their execution. These optimization will change the architecture and implementation of the design to improve performance or
to reduce cost, however, the design’s behavior will not change. For example, the sequential terminal behavior of two different implementations of the same PBS specified design would be indistinguishable. This is important to support PBS’s intended use in complex, multi-port system specification. These identical behaviors allow the system to specify pre-imposed constraints for any implementation.

2.6.5 Action Precedence

Often, several actions will be required to execute in the same cycle due to the specified behavior. When several actions are deemed to execute within a cycle, the conceptual ordering of their execution within the cycle is important. For example, consider the actions “[ x := 0; ]”, and “[ x := x + 1; ]”. If these actions execute in the same cycle, the order of execution matters. One case results in x being cleared, and in the other in x being set to 1. Thus, it is important that the PBS behavior model define multiple action execution.

Action precedence determines the conceptual ordering of actions executing within the same clock cycle. The execution of the ordered actions occurs, conceptually, in a procedural step-by-step fashion, even though in the final circuit implementation, the action execution is resolved by the data-path logic. The action precedence is determined from the production structure which enforces a partial ordering on the actions. The ordering of the actions therefore must satisfy this partial order. Thus, there are actions whose execution ordering is not defined or important, and actions whose execution ordering is determined by the production structure.

Action precedence stems from the refinement of simultaneously accepting productions. This concept is best illustrated with an example. Consider the following PBS fragment of two productions:

```
... block -> word^8; [ x := 0; ]
word -> bit^32; [ x := x + 1; ]
...
```

In this example, every time a word is recognized, the variable x is incremented. When a block is recognized, however, both of the actions are executed in the accepting cycle, since the recognition of the block occurs synchronously with the recognition of the last word. In general, several productions may be recognized simultaneously.

The rules for action precedence state that actions associated with more primitive productions and compositions execute before actions associated with more advanced productions and compositions. A production is more advanced than the productions it references. A composition is more advanced than its sub-compositions. Since the productions are not recursive, the transitivity of these statements determine the partial ordering. Actions that are unordered do not have a defined execution order. Therefore, in the example, the net
result is that \( x \) is set to zero, since the increment action is more primitive and conceptually executes first. The designer can make use of the action ordering in design. For example, actions can supplant others.

Due to underlying assumptions and how the output VHDL is structured by the PBS compiler, there is no action precedence defined between two actions of different types (combinational process and sequential process). Action precedence enforces the partial ordering only among the actions of the same type.

2.7 Special Symbols

In addition to the input interface signals, these special symbols are used as atomic terminal symbols in the productions.

2.7.1 Always True Token “.”

The special symbol “.” is used to represent the Boolean function that is always true. This symbol is used as a token that is always accepted. There are two other alternative symbols for the always true Boolean function: “‘\texttt{TRUE}’” and “‘1’”. These are equivalent to “.” and may be used interchangeably.

2.7.2 Epsilon Symbol “@”

The epsilon symbol “@” represents “nothing”, the empty token. For example the following two productions are equivalent:

\[
\begin{align*}
  p & \rightarrow a, @, b; \\
  p & \rightarrow a, b;
\end{align*}
\]

This operator is useful in several ways. It can be used as an operand in a sequential or composition to describe the case where a valid alternative may be the recognition of nothing. For example:

\[
data\_block \rightarrow \text{part1, (part2 || @), part3;}
\]

This production states that a valid \texttt{data\_block} is arranged in three parts where the second part, \texttt{part2}, may be completely absent.

Actions can be attached to the “@” symbol, and this can be convenient. Consider the following production:

\[
r\_packet \rightarrow @ \{ \text{strobe<=’0’}, \text{packet} \{ \text{strobe<=’1’}; \} ;
\]

The \texttt{r\_packet} production sets the \texttt{strobe} signal high when a \texttt{packet} is recognized (received). The designer requires that \texttt{strobe} must be low before and during the trans-
mission of the packet. Here, it is useful to attach the action to an “@” symbol prior to the packet symbol. There may be many references to the r_packet production from many different contexts of the protocol, and it would be inconvenient or difficult to specify actions that set the strobe low for all the contexts that “lead” to the r_packet sub-machine. In this case, this is done automatically, as the action is effectively attached to the excitation that initiates the packet production and “@” takes effect immediately. Thus, the “@” symbol can be used to “setup” data-flow conditions prior to the initiation of sub-machines. The symbol “’NIL’” may also be used for the epsilon symbol. The epsilon symbol is only used as a token in sequential compositions, and cannot be used in Boolean productions or Boolean compositions.

2.8 Comments

Comments may appear anywhere in the PBS specification. Text starting with “//” (as in C++) or with “--” (as in VHDL) and running until the end of the line is considered a comment. The comment text is stripped out and ignored. Exceptions to this occur when comments appear inside text encapsulations which are action clauses (both “{...}” and “[...]” types) and in drop-in segments. In these cases, the encapsulated text is extracted and used “as-is” and comments are not stripped. Comments that appear in encapsulated text segments remain until the text is eventually processed. For example, in VHDL action clauses and VHDL drop-in encapsulations, correct VHDL style comments can appear. Examples:

```
p-> foo*, bar;       // this is a comment
q -> baz^5;         -- this is also a comment
prod1 -> grok [ z := z + 1; -- increment z
                --(VHDL comment passed)
                || glop
            ]
prod2 -> zap+; { w <= 0; // clear w
                // error later! - why?
            }
```

2.9 Other Parts of the Production Based Specification

A goal for the PBS format is the specification of most, if not all, of the design information in one file. This reduces the potential side effects that can occur when making changes to the specification information. It also addresses “reproducibility of results” issues. This section describes the other portions of the PBS file which help to achieve this goal.
2.9.1 Environment and Support Section

This section contains various information about the design interface, supporting VHDL code, construction and compilation directives, and declaration that support action clauses. This supporting section consists of a collection of “drop-in” text encapsulations that may come in any order. The format for each of these drop-in’s is as follows:

\[
\text{drop_in_identifier \{ \ldots text\ldots \}}
\]

The environment and support section terminates with a “::”. The different allowed drop-in’s are described next.

2.9.1.1 Linking VHDL packages: packages {...}

This drop-in is placed at the head of the VHDL output file and is intended to be a list of VHDL libraries and packages that the compiled VHDL file will use. This drop-in is optional in VHDL. For example:

\[
\text{packages \{}
\text{library ieee;}
\text{use ieee.std_logic_1164.all;}
\text{\}}
\]

2.9.1.2 The Interface: port {...}

This specifies the interface port description of the design entity that will be used in the compiled VHDL output. This drop-in is mandatory for VHDL, but the PBS compiler will work with or without it. The following is an example port drop-in for the mouse example:

\[
\text{port \{}
\text{xc, xd, clock, reset : in bit;}
\text{xp : out integer 0 to 255}
\text{\}}
\]

2.9.1.3 Clocking: clock {...}

This drop-in specifies the clocking mechanism for the design and is used in all sequential processes in the VHDL architecture describing the compiled design entity. This drop-in is mandatory in VHDL. For example, in the mouse example:

\[
\text{clock \{ wait until clock’event and clock=’1’; \}}
\]
describes a positive edge triggered clock, and
clock { wait until (phi1'event and phi1='1') or 
(phi2'event and phi2='1'))

describes two phase clocking

2.9.1.4 Action Declarations: action_decl {...}

This drop-in is used to declare variables and signals that support the sequential actions
(“[...]”). These variable and signals are defined in the scope of the sequential action
process.

2.9.1.5 Combinational Action Defaults: combo_defaults {...}

This drop-in is used to set defaults for variables and signals which are used in the combi-
nations actions (“{...}”). These defaults are placed at the beginning of the combina-
tional action process in the VHDL.

2.9.1.6 Architecture Declarations: architecture_decl {...}

This drop-in is used to declare signals and types that are global to the entire architecture. For example, in the mouse example:

architecture_decl {
    signal x : integer range 0 to 255;
}

2.9.1.7 Architecture Insert: architecture_front {...}

This drop-in is used to wire together signals, instantiate components, define additional
processes, etc. in the front of the pbs_behavior architecture body. For example, in the
mouse:

architecture_front {
    xp <= x;
    pbs_reset <= reset;
}

See information about specifying reset conditions discussed later.

2.9.1.8 Architecture Insert: architecture_end {...}

This is similar to the previous drop-in, however, it is used to include information which
needs to be placed at the end of the architecture body.
2.9.1.9 VHDL Signal Type: bit_type {...}

All PBS generated signals default to a signal type:

```
bit        or        bit_vector(...)
```

Adding this drop-in will replace the word `bit` with the text in the drop-in, ignoring all spaces, carriage returns, tabs, and semi-colons. A common replacement for the “bit” signal type is “std_logic”.

2.9.1.10 Machine Model Flag: mealy{}

When a design is compiled, the PBS compiler, by default, builds a Moore architecture. In this architecture, the assertion of action register transfer data-flows is only a function of the state of the machine. Consider the recognition of a production with an associated action. In this Moore model, when the production is recognized, the machine transitions to a new state and then action is computed. The results of the action will not be latched into registers (and thus available) until the next state. This type of design is fully systolic.

The `mealy{}` drop-in is not really a drop-in text segment but is simply a flag. If it is present, the PBS compiler symbolically converts the Moore architecture to a Mealy architecture. In this converted Mealy architecture, the state transition and the computation of the actions occur in parallel. The assertion of action register transfer data-flows is a function of both the state of the machine and the input interface signals. When the production, mentioned above, is recognized, the transition to a new state and the storage of the results of the action computation occur in parallel. The results of the action are available for the next computation in the new cycle. Thus, in the Moore architecture, results of the action lag by a cycle and because of this, the Mealy{} architecture model is more popular. This type of design is called semi-systolic. If a Moore machine is desired, the Moore {} drop-in is optional.

2.9.2 Describing the Reset Conditions

Specifying reset conditions is done by “hooking” the predefined architecture signal “pbs_reset”. This signal is defined in the architecture declaration and is used to specify specific reset conditions. This signal is active high. If pbs_reset is set to ‘1’, the design is reset on the next clock transition. This reset behavior is synchronous\(^1\). Setting up the reset conditions can be done in the architecture_front drop-in. For example:

```
architecture_front {
  ...
  pbs_reset <= not reset;
  ...
```

\(^1\) In the future, the ability to specify asynchronous reset conditions will be added.
Here, $\text{reset}$ is active low.

### 2.9.3 Interface Signal Declaration Section

In this section, the input signals from the interface of the entity to be used in the productions are listed. The signals must be of the type specified in the “bit_type” drop-in, or if no “bit_type” is specified, they must be of the default type bit. The VHDL identifier names for signals are listed in any order, as in this example:

```vhdl
data x0 m1 m2 m3 x1
```

The VHDL identifier names for the input signals, the identifier names for the same signals in the declaration list, and their names when referenced by the productions must match. Note that, unlike VHDL, PBS is case sensitive. The declaration list and this PBS section is terminated with “::” symbol.

### 2.9.4 The Last Section

The production definition section, which was already described in detail, is the third and last section of the PBS. It contains the collection of productions. The top production must be the top-level production representing the whole design. This collection of productions is terminated by the “::” symbol as in the other two sections.

### 3.0 Running the PBS Compiler

#### 3.1 Overview - Design Flow

The designer creates a PBS specification describing the design entity using a text editor. Alternatively, another tool could generate a PBS specification as its output. An example of this is a graphical entry system or front-end tool. Each PBS file describes the behavior of one design entity. These design entities are likely components of a larger system, in accordance with the VHDL design methodology. The PBS compiler reads and processes one PBS file at a time. A hardware architecture for the design entity is synthesized by the PBS compiler during compilation and optimization stages. This architecture is reflected in the output of the tool which is a VHDL entity specification describing the architecture of the design entity at the procedural-behavioral level. This output can also be considered a RTL specification of the design architecture. After this compilation and synthesis process, the VHDL output specifying the RTL behavior of the design entity can be simulated interacting with other components using a VHDL simulation package.
The VHDL output describing the design entity is also tailored for direct hardware synthesis using the Synopsys® VHDL and logic synthesis tools. These tools compile the VHDL description of the architecture and synthesize an optimized gate-level architecture using RTL optimization and logic synthesis technology. Specific performance (timing) and area constraints and goals can be set to direct this gate-level synthesis. These detailed constraints can be specified in the PBS and passed through, in the VHDL, to the Synopsys tools. Figure 2 summarizes the design flow using the PBS compiler.

3.2 Input and Output Files

The input PBS file name ends with the extension “.pbs”. The portion of the file name before the “.pbs” is used as the name of the design entity.

The PBS compiler generates two standard files after compilation: a data file and a VHDL file. The data file contains information about the compilation and the compiled architecture. The data file’s file name consists of the design entity name followed by “.data”. The VHDL file contains the description of the RTL architecture for the compiled design. The VHDL file’s file name is the name of the design entity followed by “.vhd”.

Figure 2.
There are also three optional output files which can be generated by the compiler (see Command Line Syntax and Options in the next section). Two formats, ATT netlist and BLIF can be generated for the controller alone. In these formats, the state machines are constructed, but the actions (appearing in brackets, [...], or braces, {...}, in the PBS file) are ignored. The third format, which is currently experimental, is standard C code. In this case, the entire machine is constructed, like the VHDL output, but since the drop-ins and actions from the PBS are not parsed, they must be compatible with standard C code, rather than VHDL.

### 3.3 Command Line Syntax and Options

The PBS compiler is invoked from the command line with the following command syntax:

```
pbs { -L -r# -a -R -s -i -b -A -c -v -m -2 -d -G } pbs_file
```

The `pbs_file` is the name of the PBS file and includes the extension “.pbs”. The command line options are described below:

- `-L` don’t check design for problems (lint)
- `-r#` compute reachable states using # partitions (0 disables)
- `-a` don’t compute action relation
- `-R` don’t enable resource sharing for VHDL
- `-s` don’t use reachable states to simplify controller
- `-i` enter inspect mode after compiling
- `-b` generate BLIF (controller only)
- `-A` generate ATT netlist (controller only)
- `-c` generate C code (experimental)
- `-v` enable verbose info
- `-m` print memory stats
- `-2` force all sequences to have length 2 (for debugging)
- `-d` enable debug info
- `-G#` garbage collect at # nodes (default 100K)

Running “`pbs  pbs_file`” with no options specified will perform a reachable state analysis on the machine (without partitioning the machine) and then use that analysis to attempt to simplify the controller logic, help compute an action conflict relation, and aid in a design check. Thus, the reachable state analysis is quite useful.

The default is to generate a single machine entity (one partition) for the reachable state analysis. Unfortunately very complex designs may make the analysis difficult to complete. In those cases, there are two possibilities. First, the reachable state analysis can be turned off completely with the “`-r0`” option. Second, it can be attempted with a partitioned machine “`-r#`” option, where # is the number of partitions to create. theoretically, if
machine M2 contains 1/n as many control points (state bits) as machine M2, and machine m1 can complete the reachable state analysis in time T, then M2’s reachable state analysis will only take time T1/n to complete. Thus rarely are more than a few partitions required. The partitioning algorithm a heuristic, and will always yield a reachable state set with at least as many states as the exact solution. The controller will behave identically, but may be more complicated than necessary.

### 3.3.1 Resource sharing

Unless the -R or -a option is specified, the PBS compiler will use an if-then-else structure to explicitly show (in the VHDL output file) what combinations of actions can occur at the same time (are conflicting) or may be able to share resources, such as adders, comparators, etc. For example, with resource sharing enabled, the compiled VHDL for better_mouse.pbs (see Appendix A and B) shows that the adder and subtracter can never be used at the same time, and could potentially share one adder/subtractor:

```vhdl
if pbs_action_1 = '1' then
  x <= x - 1;
  --(PBS line 31)
else
  if pbs_action_0 = '1' then
    x <= x + 1;
    --(PBS line 30)
  end if;
end if;
```

Without resource sharing, each “if pbs_action_...” statement appears at the top level, and though some actions may never conflict, synthesis with resource sharing is much more difficult to identify.

### 4.0 Output Formats

#### 4.1 VHDL Output

The VHDL output description of the compiled architecture is a VHDL entity-architecture containing up to four VHDL processes. Combinational and sequential processes describe the behavior of the core of the machine, and potentially, two other processes describe the register transfers required by the action data-flow. The VHDL architecture view for the entity is labeled “pbs_behavior”. The core combinational process describes all of the logic associated with computing the excitation of the next state of the machine from current state variables and inputs, as well as the logic controlling activation of actions. A companion core sequential VHDL process updates the machine state on clock transitions. The register transfer structure implementing the sequential process action data-flow (the
“[ . . . ]” actions) is described in another sequential VHDL process. If there are no such actions, this process is absent. Similarly, a combinational VHDL process implements the register transfers required by the combinational process actions (the “{ . . . }” actions). This process may be absent, if there are no combinational process actions.

The VHDL output indicates line numbers where action register transfers originate in the PBS, and denotes where drop-in’s are used. This feedback of information is provided to aid the designer in inspecting the output VHDL code.

### 4.2 BLIF Output

The header of the BLIF file lists the inputs and outputs of the machine. Inputs are strictly limited to signals specified in the second section of the PBS input file. One output exists for every action in the final compiled machine. These actions appear as variables of the form “pbs.a.0”, “pbs.a.1” etc., for action zero and action one, and are ordered according to the partial ordering discussed earlier in the section *Action Precedence*.

Following the header is a list of the sequential elements. One latch is allocated for every control point (state bit) to separate the present and next state variables.

Next follows a list of logic functions for the intermediate variables. Conceptually, each logic function is a multiplexer, in which the first variable in the list selects between the next two. Each logic equation is a function of intermediate variables, present state bits, and inputs. For example:

```fortran
.names pbs.x.8 t0x16421c t0x71108 t0x70254
11- 1
0-1 1
```

Finally, there is a list of assignments of next state variables and outputs to the intermediate variables used in the list of logic functions.

The BLIF output format is intended to be used for logic minimization of the controller, not as a final, directly synthesizable machine.

### 4.3 ATT Netlist Output

The ATT netlist output file organization is quite similar to the BLIF output. The header indicates the name of the entity, given by the name of the PBS input file, followed by a list of inputs and outputs of the circuit. The list of inputs include those specified in the second section of the PBS input file, and a clock, “CK”. The list of outputs consists of all of the outputs of the state machine, one for each action, ordered according to the partial ordering discussed above.
Next, the file has a list of memory elements used for the next state variables.

Finally there is a list of gates which describes the next state variables, output functions, and intermediate variables in terms of present state, inputs, and intermediate variables. Note that while there may be some optimization, including sharing, or fan-out of intermediate variables, and simplification based on the reachable state analysis, the ATT netlist output is intended to be used for logic minimization of the controller, rather than a final, directly synthesizable machine description.

4.4 C Code Output

The C code output is currently experimental. Like the VHDL output, the C output file represents the entire machine, including actions. However, as mentioned earlier, a PBS file that generates VHDL output will need to be modified to generate correct C output. Drop-in’s and actions from the PBS file are not parsed, and so are assumed to be written in the target output language.

5.0 Debugging the Design

5.1 Inspect Mode

*Inspect Mode* works like a tiny simulator. The designer can stimulate and test the behavior of the compiled design for inspection. Design characteristics can be checked and behavior observed. When actions are “fired,” they are not really executed or interpreted, but they are displayed symbolically in order of their execution, along with their corresponding line numbers in the original PBS. When inspect mode is entered, the design is reset and a command prompt is displayed. The machine’s reset state is the following: the machine state is set to the start value, and all of the input interface signals are set low. At this point, commands can be entered to: change signal levels, print status, step the machine one cycle, reset the machine, and exit the simulator. The Inspect Mode commands are summarized in Table 3, below:

<table>
<thead>
<tr>
<th>command</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>Step and run simulator for one cycle.</td>
</tr>
<tr>
<td>p</td>
<td>Print status and state.</td>
</tr>
<tr>
<td>h &lt;i₁&gt; &lt;i₂&gt; .. &lt;iₙ&gt;</td>
<td>Set interface signals &lt;i₁&gt; &lt;i₂&gt; .. &lt;iₙ&gt; high.</td>
</tr>
<tr>
<td>l &lt;i₁&gt; &lt;i₂&gt; .. &lt;iₙ&gt;</td>
<td>Set interface signals &lt;i₁&gt; &lt;i₂&gt; .. &lt;iₙ&gt; low.</td>
</tr>
<tr>
<td>r</td>
<td>Reset the machine to the start state.</td>
</tr>
<tr>
<td>q</td>
<td>Quit the simulator.</td>
</tr>
</tbody>
</table>
For large simulations, inspect mode simulation commands can be entered into a text file and the input to the PBS compiler can be redirected from standard input to read this file. Inspect Mode is very useful for design debugging and inspection, however, it should not be used as a substitute for complete simulation with a VHDL simulator.

5.2 Simulation

VHDL simulation of the design architecture interacting with other design entities is necessary to validate the design. It may also be more efficient in tracking down bugs than using inspect mode. VHDL test benches may be more efficient in exercising specific portions of the protocol that would take a long time to set-up and simulate using Inspect Mode, especially if the interaction is between other complex components. Gate level simulations of the synthesized design provide simulations that more closely reflect the switch-level time behavior of the design.

5.3 Synthesis

Sometimes, synthesis of the design can provide clues that aid in debugging. For example, inspection of the circuit for structures present or absent, or feedback can give clues as to the nature of some bugs. If there is confusion regarding how production constructs or compositions work, it is sometimes helpful to isolate the appropriate portions in small examples. Inspection of VHDL output or the circuit after synthesis is also helpful.

6.0 About the PBS Compiler

Version 2.2 of the PBS compiler is coded in C++. The PBS specific code is comprised of approximately 11,400 total lines. In addition, there exists about 27,000 lines of generic code for implementing reusable classes including a C++ BDD package.

7.0 References

Appendix A: Complete Design Examples

--------------------- better_mouse.pbs ---------------------

port {
    xc, xd, clock, reset : in bit;
    xp : out integer range 0 to 255
}

clock { wait until clock'event and clock = '1'; }

architecture_decl {
    signal x : integer range 0 to 255;
}

architecture_front {
    xp <= x;
    pbs_reset <= reset;
}

mealy {}

:: xc xd ::

::

mouse -> .*, event;
  event -> forward || reverse;
  forward -> (xd):rising;  [ x <= x + 1; ]
  reverse -> (~xd):rising;  [ x <= x - 1; ]
  rising -> ~xc+, xc;

::
-- midi protocol machine by Egil Vassend
-- and Andrew Seawright
-- converted to pbs v2.0 format - A.S.
--

port {
    clock : in BIT;
    data : in BIT;
    reset : in BIT;
    strobe : out BIT;
    midi_cmd : out BIT_VECTOR (1 downto 0);
    data_R1 : out BIT_VECTOR (5 downto 0);
    data_R2 : out BIT_VECTOR (6 downto 0)
}

architecture_front { pbs_reset <= reset; }

clock { wait until clock’event and clock = ’1’; }

action_decl {
    variable R1 : BIT_VECTOR (5 downto 0);
    variable R2 : BIT_VECTOR (6 downto 0);
}

combo_defaults { strobe <= ’0’; }

mealy {}

::
data

::

::

::

top -> (event!R)*;
event -> action { strobe <= ’1’; }
    || ignore;
action -> note_off
    || note_on
    || program_change
    || all_notes_off
    ;

note_off -> note_off_cmd, data_freq_adr, data_ampl_off;
    [
        midi_cmd <= "01";
        data_R1 <= R1;
        data_R2 <= "0000000";
    ]

note_off_cmd -> idle, startbit, ONE, ZERO^3, ANY^4, stopbit;

note_on -> note_on_cmd, data_freq_adr, data_ampl_on;
    [
        midi_cmd <= "01";
        data_R1 <= R1;
        data_R2 <= R2;
    ]

note_on_cmd -> idle, startbit,
    ONE, ZERO^2, ONE, ANY^4, stopbit;

data_freq_adr -> idle, startbit [ R1 := "000000";],
    ZERO, ANY, databit_A^6, stopbit;

data_ampl_off -> idle, startbit, ZERO, ANY^7, stopbit;

data_ampl_on -> idle, startbit [ R2 := "0000000";],
    ZERO, databit_B^7, stopbit;

program_change -> program_change_cmd, data_mod_ampl;
    [
        midi_cmd <= "10";
        data_R2 <= R2;
    ]

program_change_cmd -> idle, startbit,
    ONE^2, ZERO^2, ANY^4, stopbit;

data_mod_ampl -> idle, startbit [ R2 := "0000000";],
    ZERO, databit_B^7, stopbit;

all_notes_off -> all_notes_off_cmd, data_byte1, data_byte1;
[ 
    midi_cmd <= "11";
    data_R1 <= "000000";
    data_R2 <= "0000000";
]

all_notes_off_cmd -> idle, startbit,
                    ONE, ZERO, ONE^2, ANY^4, stopbit;

data_byte1 -> idle, startbit, ZERO, ANY^7, stopbit;

ignore -> idle, startbit, ONE^4, ANY^4, stopbit
         || idle, startbit, ONE^2, ZERO, ONE, ANY^4, stopbit
         || idle, startbit, ONE, ANY, ONE, ZERO, ANY^4, stopbit ;

data_bit_A -> data_one_A || data_zero_A;

data_bit_B -> data_one_B || data_zero_B;

data_one_A -> ONE; [ R1 := R1(4 downto 0) & '1'; ]
data_zero_A -> ZERO; [ R1 := R1(4 downto 0) & '0'; ]
data_one_B -> ONE; [ R2 := R2(5 downto 0) & '1'; ]
data_zero_B -> ZERO; [ R2 := R2(5 downto 0) & '0'; ]
idle -> ONE*;

startbit -> ZERO;
stopbit -> ONE;

ONE -> data;
ZERO -> ~data;
ANY -> 'TRUE';

::
Appendix B: VHDL Output Example

-- PBS Version v2.1 - Tue Aug 30 16:54:17 PDT 1994
--
-- PBS Generated VHDL entity
--
-- PBS Compiler Copyright (c) 1992,1993
-- University of California.
--
--

entity better_mouse is
port (xc, xd, clock, reset : in bit;
     xp : out integer range 0 to 255
      );
end;

architecture pbs_behavior of better_mouse is
begin

    signal pbs_X, pbs_Y : bit_vector (0 to 7);
signal pbs_reset : bit;
signal pbs_action_0 : bit;
signal pbs_action_1 : bit;

    signal x : integer range 0 to 255;

    xp <= x;
pbs_reset <= reset;

begin

    pbs_main_combo : process (pbs_X, xd, xc
variable t0x53ffc, t0x53f24, t0x542b4, t0x5429c, 
t0x542e4, t0x146994, t0x14697c, t0x146964, 
t0x542cc, t0x54014, t0x53f0c, t0x5402c, 
t0x54044, t0x14694c : bit;
begin

begin

t0x53f24 := not xc; t0x53ffc := xd and t0x53f24;
t0x5429c := pbs_X(2) and t0x53ffc;
t0x542b4 := (pbs_X(3) and t0x53ffc) or ((not pbs_X(3)) and 
t0x5429c);
t0x53f0c := xc;
t0x54014 := xd and t0x53f0c;
t0x542cc := pbs_X(2) and t0x54014;
t0x542e4 := (pbs_X(3) and t0x54014) or ((not pbs_X(3)) and 
t0x542cc);
t0x5402c := (not xd) and t0x53f24;
t0x14694c := pbs_X(5) and t0x5402c;
t0x146964 := (pbs_X(6) and t0x5402c) or ((not pbs_X(6)) and 
t0x14694c);
t0x54044 := (not xd) and t0x53f0c;
t0x14697c := pbs_X(5) and t0x54044;
t0x146994 := (pbs_X(6) and t0x54044) or ((not pbs_X(6)) and 
t0x14697c);

pbs_Y(0) <= '0';
pbs_Y(1) <= '1';
pbs_Y(2) <= t0x53ffc;
pbs_Y(3) <= t0x542b4;
pbs_Y(4) <= t0x542e4;
pbs_Y(5) <= t0x5402c;
pbs_Y(6) <= t0x146964;
pbs_Y(7) <= t0x146994;
pbs_action_0 <= t0x542e4;
pbs_action_1 <= t0x146994;

end process;

-----------------------------------------------------------

pbs_main_sequential : process
begin
----- dropin start -------------------------------------
wait until clock'event and clock = '1';
----- dropin end ---------------------------------------

if pbs_reset = '1' then
    pbs_X(0) <= '1';
    for i in 1 to pbs_X'right loop
        pbs_X(i) <= '0';
    end loop;
else
    pbs_X <= pbs_Y;
end if;
end process;

-----------------------------------------------------------

pbs_action_sequential : process begin

----- dropin start -------------------------------------
wait until clock'event and clock = '1';
----- dropin end ---------------------------------------

if pbs_reset = '1' then
    null;
else
    if pbs_action_1 = '1' then
        x <= x - 1;
        --(PBS line 31)
    else
        if pbs_action_0 = '1' then
            x <= x + 1;
            --(PBS line 30)
        end if;
    end if;
end if;
end if;
end process;

-----------------------------------------------------------

end pbs_behavior;
Symbols

! sequential not operator 14
!! exception operator 15
!E excitation operator 16
!R reset operator 15
& Boolean AND 16
&& sequential AND operator 13
* Kleene Closure operator 13
+ one or more operator 14
, concatenation operator 11
: always TRUE token 22
:1 6
: qualification operator 16
@ empty token 22
* Boolean XOR 16
* multiple concatenation operator 12
  Boolean OR 16
|| sequential OR operator 12
~ Boolean NOT 16

A
action clauses 19
  acceptable placement 20
action_decl 25
actions 8, 18, 19
declarations 25
execution 21
precedence 21, 22
architecture
declarations 25
architecture_decl 25
architecture_end 25
architecture_front 25
ATT netlist 29, 31

B
behavior
  PBS design entity 20
time sequence 9
bit_type 26
BLIF 29, 31
Boolean AND 16
Boolean EXCLUSIVE-OR 16
Boolean NOT 16
Boolean OR 16

C
C code 32
clock 24
combo_defaults 25
command line syntax 29
comments
  PBS 23
  BLIF 29, 31
concatenation operator 11

D
declaration 6
design interface 6, 24
drop-ins 24, 32

E
environment and support section 24

F
files
  input 28
  output
    ATT netlist 29, 31
    BLIF 29, 31
    C code 29, 32
data 28
target language 32
  VHDL 28, 30

G
garbage collection 29

H
handler 15
hooking 26

I
input signals 6
Inspect Mode 32
  commands 32
interface
  VHDL 24
  interface signal declaration section 27

K
Kleene Closure operator 13

L
logic synthesis 28

M
packages
  PBS 23
  VHDL 24
  PBS compiler 27
PBS declarations 6
pbs_reset 26
port declaration 24
processes
  combinational 19
  sequential 19
production definition section 27
productions 6, 7, 10
  advanced 21
  Boolean 7
  example 8, 10, 13
  primitive 21
  recursive 9
  referencing 9, 10
  sequential 7
  symbols 9
  syntax 9
  top-level 8

qualification operator 16

reachable state analysis 29
  partitioning 30
  partitions 29
  time complexity 30
recovery 15
reset conditions 26
reset operator 15
resource sharing 29
resources
  conflicts 30
  sharing 30
re-synchronization mechanism 15

sequence factoring 12
sequential and operator 13
sequential NOT operator 14
sequential OR operator 12
signal type
  VHDL 26
simulation
  PBS 32
  VHDL 33
specification
  applicative 6
  declarative 6
symbols 9
  special 22
  terminal 22
synthesis 33

tokens 7
  recognition 8

VHDL 6, 29, 30
  combinational process 19
  sequential process 19