Synthesis from Production-Based Specifications

Andrew Seawright and Forrest Brewer
Department of Electrical and Computer Engineering
University of California, Santa Barbara, CA 93106

Abstract

This paper describes a model for, and an implementation of, production-based synthesis of hardware description language (HDL) code in which the overall structure of the resultant machine is derived from a hierarchy of sub-machine descriptions, each represented by a production. The production-based specification (PBS) consists of productions annotated with HDL action code, and forms the input to a design tool which outputs procedural HDL tailored for hardware synthesis. Due to the concise nature of this form of specification, for certain machines such as protocol controllers, the technique can save enormous labor in the construction of procedural specifications for these machines. Novel aspects of this research include the compilation of a PBS with HDL action clauses into synthesizable procedural HDL and the approach to specification of machine behavior in the event of exceptional conditions.

1.0 Introduction

A common way to describe the behavior of a machine is in a procedural form using a hardware description language (HDL) such as VHDL [13]. This type of description is procedural in the sense that the designer describes the behavior in a step by step sequence. An example procedural specification is shown in Figure 1, which describes a

```
process
begin
wait until CLK'event and CLK = '1';
if(SEEN_TRAILING and DATA = '0') then
   IS_LEGAL <= '0';
   COUNT <= 0;
elsif(SEEN_ZERO and DATA = '1') then
   SEEN_TRAILING := TRUE;
elsif(DATA = '0') then
   SEEN_ZERO <= '1';
   COUNT := COUNT + 1;
end if;
end process;
```

Figure 1

machine that counts consecutive zero bits in a sequential single bit input stream. Simulation and direct hardware synthesis is commonly performed from procedural descriptions. An alternative description is shown in Figure 2.

```
Count -> Valid | Invalid;
Valid -> ONE* Low* ONE*;
{
     IS_LEGAL <= '1';
}
Invalid ->ONE* ZERO+ ONE+ ZERO;
{
     IS_LEGAL <= '0';
     COUNT := 0;
}
Low -> ZERO; { COUNT := COUNT + 1;}
```

Figure 2

This second type of description is based on productions. A production is a rule describing certain compositions of symbols as a single symbol of higher abstraction. Symbols at the lowest level of abstraction (tokens) describe atomic sets of external signal transitions, while those at higher levels describe arbitrarily complex sequences of these tokens. In the example above, ONE and ZERO are tokens. Note that some productions initiate actions which are fragments of HDL code enclosed in braces.

For many types of complex behavior, production based specifications (PBS) are superior to procedural specifications, in that they can be more concise, and easier to debug and understand. Examples include communications protocols, cache protocols, bus controlers, adaptive coding, and translators. The simplicity of interpretation stems from the local nature of each production in the specification. Each production is simultaneously active for all input transitions so that the designer need not worry about the explicit construction of the global control flow, which is necessary in designing a procedural specification.

The synthesis procedure described in this work converts a production specification of machine behavior into a procedural description encoded in a specified hardware description language and tailored explicitly for hardware synthesis. This output description contains explicitly the minimal controlling FSM described implicitly by the productions. The advantage of the HDL output description is that it enables fast turnaround simulation or synthesis of the machine using state-of-the-art HDL tools. Since for many applications the production hierarchy is very concise, the resulting HDL has a greater chance of representing the designer's intent than a hand coded procedural

description. In this respect, the user model for this technique is a hardware analog of the very successful compiler generation tools prevalent in software design [3] [4] [8].

2.0 Related Work

The related software tools compile grammar productions and pattern recognition expressions into parsers, and lexical scanner routines. Many of the advantages present in using these software tools *must be* applicable to the specification of control dominated behavior. However, since the trade-offs involved in designing a hardware machine are different than for generating a software program, the models, metrics, and techniques used in compiling these specifications differ substantially. With respect to hardware, the most closely related work is due to Ullman et. al. [1] [2] in their study of the compilation of regular expressions into PLA implementations. In this work, a single regular expression is compiled into a non-deterministic finite automata (NFA). This NFA was directly implemented as a PLA, with combinational feedback terms. The non-determinism of the current state in the machine was represented by these feedback terms, each representing a single active state. Verification of equivalent behavior of differing implementations of machines was done using automata based models [5]. Their model for representing classes of equivalent behavior is in some respects similar to the model presented in this paper representing the target machine's behavior after the production description is compiled.

Due to the fact that the PBS describes a hardware entity, a multitude of CAD techniques and technologies are available to create an efficient implementation. These techniques include: 1) The synthesis of logic networks from procedural hardware description languages such as implemented in the BDSYN tool [6] and industrial CAD packages [14]. Of particular importance to this technique is the notion of resource allocation and sharing among the several possibly mutually exclusive operations implied in the action clauses in the PBS. 2) The optimization of logic networks to meet timing and area constraints offered by logic synthesis [6] [7]. 3) The techniques of state assignment [11] [12] and minimization [10] of finite state machines.

3.0 Methodology and Advantages

In use, the tool provides an HDL generator. The output VHDL can be both simulated and synthesized. Desired changes in the specification are made by constructively modifying the PBS description and re-compiling. This technique has several advantages: 1) Modification of a design can be readily made, as the productions form a natural partitioning of the machine's behavior. 2) The compilation and optimizations of the productions are inherently symbolic, enabling the designer to specify behavior at many levels of abstraction, from single bit events to valid protocol recognition. 3) Powerful control and data flow analysis may be possible from the hierarchical PBS description. 4) After synthesis the structural architecture can be simulated and compared with the procedural simulation in order to track design integrity. 5) The procedural HDL output can be tailored and annotated by the production based specification compiler to match a synthesis policy. This allows customization of the output for direct logic synthesis using currently available tools.

4.0 Model

A Mealy [9] machine is defined as a 6-Tuple (Q, S, D, d, l, q0), where Q is a finite set of states, S the input alphabet, and D the output alphabet. The transition function d maps Q x S to Q, while l maps Q x S to D. The initial state of the machine is q0. Our goal is to design an entity M, which is essentially an FSM when viewed at the interface level, but could in fact be a complex controller with arbitrary datapath or other interior extensions. We model M as an 8-Tuple (Q, S, A, d, L, q0, I, X) where Q, S, d, and q0 are the same as in the Mealy machine above. L is a function of domain Q x S mapping to a range consisting of arbitrary ordered subsets of A. The elements of A are "actions" performed by the FSM under certain state transitions. These actions are segments of procedural HDL code. Unlike the Mealy machine's output function, L maps to ordered sets of these arbitrary high-level segments. I is a vector of boolean signals providing the totality of the machine's external interface. We define X as an function that maps the interface, I, to elements of the alphabet S. X provides a level of abstraction between the interface signal vector and the internal input language for the FSM and other possible uses for the interface signals (such as operands to/from the actions).

A production-based specification describing M is compiled using the PBS compiler to obtain complete register transfer level procedural description of the machine M. This RTL description consists of, among other things, the implementations of the functions X, d, and L. The PBS input consists of a set of productions, annotated with associated actions forming the set A, and auxiliary HDL code. The productions specify the behavior of the machine through the actions, as a function of internal state and the input stimulus. The terminal symbols of the productions form the set S. The auxiliary HDL code defines the interface I, the mapping function X, initialization code for the state q0, and other user defined HDL code. The user, or another tool, is free to define the specifics of the clocking strategy and the mapping X in the specification, as long as

it can be synthesized. For the rest of this paper we assume a synchronous framework.

The following statements codify our model with respect to the specification of the behavior of the machine through the action clauses:

- The behavior of the machine with respect to the external world is defined by the behavior of a machine, derived from the production specification, in which the actions are combinationally executed in the designated points of the protocol and within a single clock transition.
- When there are several actions defined during a state transition of M, the actions of more primitive productions have higher precedence and conceptually execute before the actions of more advanced productions. To clarify, we define the behavior of the machine with several simultaneous actions to be the behavior of the machine if the actions were procedurally executed in step by step order even though they may be resolved by a single combinational logic function.
- A production is more primitive if it is called by the more advanced production.

Given the above points concerning the *behavior* of the machine as defined by the specification and viewed by the designer, any transformation of the machine is certainly allowed that results in the same external behavior. Such transformations would be made to improve performance or minimize cost.

4.1 Hardware vs. Software

There are different trade-offs and implicit assumptions in specifying and constructing a finite state machine to perform some task as a hardware device, rather than as a software program, using productions. In compiling programs in software, we often have the luxury of lookahead, e.g. looking at the future input tokens to disambiguate the meaning of the input. In hardware, lookahead may be difficult if not impossible. For example, in this model, input is defined through the X function as the state of the interface I during a clock transition. In this case, lookahead is simply *not* possible (lookahead can be simulated using an X function with state). Another difference in hardware is due to the presence of rigid timing constraints. For example, between successive cycles of a clock a FSM must compute its next state based on the current input. A controller may be required to respond to an input event in the next clock cycle. Although software compilers are designed to be efficient it may be of little concern that several "cycles" of the parsing loop are required to interpret the meaning of an input token before the next token is read. This is true in the classic shift/reduce parser [8]. A further distinction is found in the meaning and mechanism for producing "tokens". In hardware, we have a wide variety of possible candidates for tokens including operands on busses and transitions of input signals. Finally, the issue of error recovery is important in both hardware and software. In program compilers, the emphasis is usually placed on identifying errors in the input stream, and terminating the compilation. On the other hand, in hardware the emphasis is on conveniently specifying well-defined behavior of the machine with respect to abnormal or exceptional input. For example an exception may require the machine to perform additional processing, resynchronization, and recovery procedures.

5.0 Implementation

A PBS compiler has been implemented, in approximately 4600 lines of C++, which compiles input specifications containing productions, VHDL actions, and VHDL code segments. The productions are composed of symbols and operators. The symbols represent sub-machines ("calls" to other productions), or terminal symbols. In the current implementation, recursion in the productions is illegal. Each symbol in a production represents an instantiation of a sub-machine that performs some task. Terminal symbols represent machines that recognize a single input symbol. During the compilation process, any symbols in the productions that are not calls to other productions, i.e. are not non-terminals, are considered terminals forming the set S. The user explicitly specifies in the PBS description explicitly how the tokens are generated; i.e. the X function.

The operators in the productions combine the sub-machines into more complex machines through compositions of the sub-machines. These operators consist of the standard regular expression operators [8] [9] [10] plus several special operators. The special '^' operator is for convenience and is used as a shorthand for representing the concatenation of several identical sub-machines by the specified number of instantiation. For example the production below illustrates its use:

```
block -> byte^4;
```

The production defines `block` to be a sequence of four `byte`s.

The exception operator, '!', is used to specify conveniently the behavior of a sub-machine in the circumstance of non-recognition. For example in the production:

```
data -> block!resync;
```

The exception operator states that if the sub-machine `block` recognizes events that can not be `block` *or any other valid production*, then the state is transferred immediately to the sub-machine represented by the `resync` production. When then `resync` production accepts, the effect is as if the `block` machine accepts. Several levels of ex-

ception handling are possible, with well defined scoping rules and precedences. The '!!' operator is another exception operator. When used, the annotated sub-machine is constructed with exceptional behavior causing a "reset" of the sub-machine.

The exception operators allow the user to specify concisely behavior in the case of events not explicitly described in the ordinary productions, by giving access to the complement space of the production. The complement space for the production is defined with respect to all of the ensemble productions. This allows a convenient mechanism for specifying re-synchronization. Of course the user is also free to explicitly add specific error productions in the specification as well.

The productions can be annotated with VHDL code segments as "actions". These actions may be associated with any sub-machine symbol in the production or with the entire production as a whole. In either case, the action is assumed to take effect on valid recognition of the input conditions as described in the model section. Consider the example machine introduced earlier. When the machine detects an invalid input frame, the last token will be a ZERO. Two actions are indicated: the action attached to the ZERO token in the `low` production, and the action associated with the `invalid` production. Due to precedence the more primitive action "computes" first. In this case the overall effect is the `COUNT` variable reset to zero.

5.1 Compilation

The productions are collapsed and compiled into an intermediate tree representation. The interior nodes of the tree represent the various composition operators, and the leaves of the tree represent the terminal token symbols. To be efficient, common sub-trees are shared when there are multiple references to a production from other productions in the PBS. Since each node of the tree represents a particular point in the protocol, action objects are attached to lists provided at the nodes. In the current implementation, the action objects consist of wrappers around VHDL code.

A non-deterministic finite automata (NFA) is constructed from this tree using a process based on Thompson's Construction [9], however suitably modified to implement the exception handling mechanism and other extensions. Actions are propagated from the tree to the NFA while maintaining their precedence. Next the NFA is converted to a DFA. This is done using the well-known subset construction algorithm [9], again with extensions for implementing the exception handling mechanism, particularly in the method in which the 'death' state is handled. The death state is a DFA state generated in the construction process when a group of NFA states do not have successor(s) for some input symbol. All of the transitions leaving the DFA death state return to itself, thus the machine can never leave this state. Exception handling is described in more detail in the next section.

The next step in the compilation process is the optimization and encoding of the of the DFA. The number of states in the DFA is minimized using a partition refinement technique [10], and the states are encoded using a heuristic based on [12]. Finally synthesizable VHDL code is generated.

5.2 Exceptions and Compilation

When a PBS containing exception operators is compiled, the exception operators mark sub-trees in the intermediate representation with a specific exception "scope". These scope regimes may be hierarchically nested and associate non-recognition in the indicated sub-tree with a specific error sub-machine (represented as another sub-tree). Each exception scope is represented by a "color". The *natural* color annotates sub-trees not contained in any exception scope; e.g. no exception was specified. When the NFA is generated, all of the NFA nodes are colored in accordance to their relationship to the tree representation.

Recall, when the DFA ia constructed, a "death" state is generated when a group of NFA states have no successor state(s) for a transition on a particular input symbol. To implement the exception scoping, *several* different death states are generated. Each death state is marked by a unique *combination* of exception colors collected from the NFA nodes implying the transition. Due to the potential for non-determinism expressed in the NFA, a DFA death state may be marked by several exception colors. In the most typical cases of exception operator use, this is not manifest. However, complex production expressions containing exception operators are certainly possible. In this case, one of the exception colors dominates. This dominating color is determined by a simple precedence rule. In the final steps, all of the colored death states, except the one annotated by the natural color, are removed from the DFA and the transitions to these states are redirected to the DFA equivalents of the exception machines (note that no actions are lost in this process since the DFA is a mealy machine and actions are associated with the transitions, not the states). The exception behavior due to the '!!' operation is handled in a similar way.

5.3 VHDL output

The PBS compiler's output is two files: a VHDL package, and a VHDL entity/architecture body. The package file contains definitions, declarations, token and state encodings, and a state transition table. The entity/architecture

body consists of the machine body with action code, and other user defined VHDL code segments “dropped” into the VHDL entity after compilation. The skeleton format of the VHDL entity body is shown in Figure 3. The main body of the machine is contained in the “machine core” section which contains the action code as well. The gray regions are locations in the skeleton where the user code segments are inserted using special directives. The directives, and the corresponding code enclosed by braces, is placed in the beginning of the PBS description, and can occur in any order. Several directives are mandatory such as “port{}” since a useful VHDL entity must have an interface. Most are optional. Figure 4. illustrates the directives required by the example.

5.4 Descriptive Partitioning Issue

We introduce this idea by noting the existence of input specifications, even quite small ones, that result in the generation of an enormous number of states in the DFA during compilation. In worst case the growth of the number of DFA states is an exponential function of the growth of the number of NFA states [8]. This arises essentially when the DFA states are used as “memory” as opposed to strictly “control state” due to the manner of specification. An example is the storage and later comparison of permutations of large numbers of past input symbols. Although extremely bad state explosion behavior certainly exists for pathological specifications, it does not detract from the utility of this specification technique. We have found that, for most types of machine behavior suited to PBS style specification, a natural partitioning of design specification is typical. This partitioning is between the description of the control aspects of the design through the productions, and the specification of data-paths, memory storage, and semantics through the HDL action clauses, and other procedural or structural entities. Storage-in-mass of input data can also be achieved via interactive behavior specified by productions, with memory storage entities.

6.0 Experiments

Several example production base specifications have been compiled and synthesized. Table 1 illustrates the results of the experiments[1] to date. The cache example is a simple cache coherency protocol from [15]. Parity is a sequential machine that recognizes only even parity bytes. It is an interesting example because it is a good benchmark for the optimization algorithms. The bounce example behaves as a bit sequential low pass filter. This example is from [1]. The count0 example is the complete description of the machine introduced earlier, and is a PBS implementation of the procedural design in [14]. The pager2 example is a large example adapted from a protocol for radio pagers, requiring complex control. The VHDL actions in this specification perform such functions as assembling data words and implementing linear feedback shift registers for error detection. The PBS and VHDL size columns are the total number of lines in the input descriptions and output descriptions respectively. Note that the VHDL output makes use of very concise jump transition tables and hand coded procedural versions of these files would probably be much larger. It should be noted that the smaller examples have fixed overhead in the form of reset conditions and interface descriptions which dominate the size of the smaller PBS descriptions. This can be seen by comparing the number of lines of productions and actions versus the total PBS description size. In the larger example (pager2), a hand coded procedural description would have to represent at least 536 distinct states, and 39 distinct actions, while this behavior can be concisely represented in 139 lines of productions. The NFA state column contains data for the total number of NFA states before conversion to a DFA. The compilation times are CPU seconds running on a Sun Sparcstation 1® workstation for compiling the PBS to procedural VHDL. This time includes state minimization, but excludes state assignment. The last two rows of the table show the results of further compilation and logic synthesis of the designs using the Synopsys® synthesis tools.

7.0 Conclusion and Acknowledgment

A model and an implementation for hardware specification, simulation, and synthesis from production based specifications has been described. For many types of complex behavior, production based specifications have advantages over procedural specifications. We believe this specification technique to be useful for the ASIC designer, as well as having potential for further research in the field of system level design. Currently we are studying optimization of the data flows implied by the HDL actions in the framework of the model presented in this work.

The authors wish to acknowledge Synopsys Inc. for valuable discussion and for the use of their VHDL compiler and logic synthesis tools in obtaining the data for this paper. This research was supported through grants from Synopsys and the California MICRO program #90-195.

1. The PBS descriptions for these examples may be obtained via anonymous ftp from bears.ece.ucsb.edu.

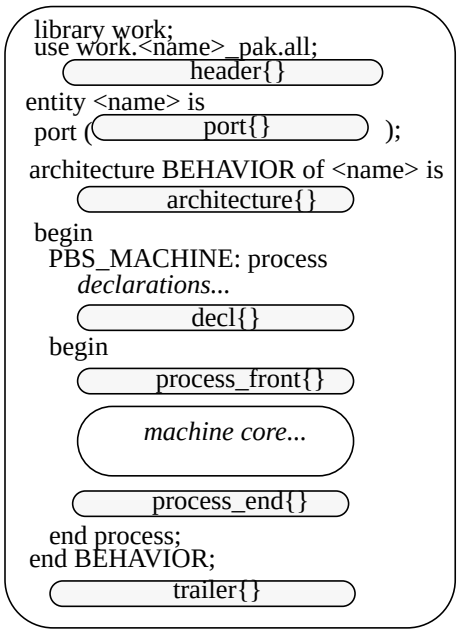

Figure 3

```
port { CLK, INPUT: in BIT;
       CNT: out INTEGER range 0 to 7;
       IS_LEGAL: out BIT}

process_front {
       wait until CLK'event and CLK = '1';
       if (input = '1')
           then PBS_TOKEN := ONE;
           else PBS_TOKEN := ZERO;
       end if;}

decl { variable COUNT: INTEGER range 0 to 7;}
```

Figure 4

Table 1: Experiments

metric	cache	parity	bounce	count0	pager2
number of productions	5	17	5	4	21
number of actions	2	2	2	3	39
lines of productions and actions	11	21	9	4	139
PBS size (lines)	41	48	36	41	187
procedural VHDL output	83	120	96	108	1269
number of NFA states	18	1020	13	30	1688
final number of states (DFA)	3	16	5	4	536
number of transitions with actions	3	4	2	6	548
compilation time	0.1	2.0	0.1	0.1	18.9
number of standard cells	9	26	14	30	***[a]
relative area	23	35	45	102	***[a]

a. hardware synthesis results not available

References

1. A. R. Karlin, H. W. Trickey, and J. D. Ullman, "Experience With A Regular Expression Compiler," Proceedings of the ICCD: VLSI in Computers, 1983.
2. R. W. Floyd, and J. D. Ullman, "The Compilation of Regular Expressions into Integrated Circuits," Journal of the Association for Computing Machinery. 29:2, 1982.
3. M. E. Lesk, "Lex - a lexical analyzer generator," Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, N. J. 1975.
4. S. C. Johnson, "Yacc: Yet Another Compiler Compiler," Computing Science Technical Report 32, AT&T Bell Laboratories, Murray, Hill, N. J. 1975.
5. S. Devadas, and K. Keutzer, "An Automata-Theoretic Approach to Behavioral Equivalence," proc. ICCAD 1990.
6. R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "MIS: A Multiple-Level Logic Optimization System," IEEE T-CAD, Vol. 6, No. 6, November 1987.
7. A. J. de Geus, and D. J. Gregory, "The Socrates Logic Synthesis and Optimization System," in Design Systems for VLSI Circuits Logic Synthesis and Silicon Compilation. Dordrecht: Martinus Nijhoff Publishers, 1987.
8. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers Principles, Techniques, and Tools.* Reading, MA: Addison-Wesley, 1986.
9. J. E. Hopcroft, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation.* Reading MA: Addison-Wesley, 1979.
10. Z. Kohavi, *Switching and Finite Automata Theory,* New York: McGraw-Hill, 1978.
11. G. De Micheli, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Optimal State Assignment for Finite State Machines," IEEE T-CAD, Vol. 4, No. 3, July 1985.
12. S. Devadas, H-K. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli, "MUSTANG: State Assignment of Finite State Machines Targeting Multilevel Logic Implementations," IEEE T-CAD, Vol. 7 No. 12. December 1988.
13. *IEEE Standard VHDL Language Reference Manual,* ANSI/ IEEE 1076-198, New York: IEEE, 1988.
14. S. Carlson, *Introduction to HDL-Based Design Using VHDL,* Mountain View: Synopsys, 1990.
15. J. L. Hennessy, and D. L. Patterson, *Computer Architecture A Quantitative Approach*, pp 469-471. Morgan Kaufmann Publishers: San Mateo 1990.