Clairvoyant: A Synthesis System for Production-Based Specification

Andrew Seawright and Forrest Brewer
(andy@synopsys.com, forrest@ece.ucsb.edu)

Abstract

This paper describes a new high-level synthesis system based on the hierarchical Production-Based Specification (PBS). Advantages of this form of specification are that the designer does not describe the control flow in terms of explicit states or control variables and that the designer does not describe a particular form of implementation. The production-based specification also separates the specification of the control aspects and data-flow aspects of the design. The control is implicitly described via the production hierarchy, while the data-flow is described explicitly in action computations. This approach is a hardware analog of popular software engineering techniques. The Clairvoyant system automatically constructs a controlling machine from the PBS and this process is not impacted by the possibly exponentially larger deterministic state space of the designs. The encodings generated by the constructions compare favorably to encodings derived using graph-based state encoding techniques in terms of logic complexity and logic depth. These construction techniques utilize recent advances in BDD techniques.

1.0 Introduction

In conventional high-level and register-transfer-level hardware description languages, the control structure of a design is typically specified using conditional language constructs such as *if-then-else* and *case* statements. Conditional branching in the control flow is determined by the evaluation of program state variables which are explicitly specified. For many problems, however, the specification of the machine behavior in this format is cumbersome. The designer may wish to work at a higher level of abstraction in which the detailed interaction of the sub-components is resolved automatically. This is especially true for problems in which the time sequence behavior is complex or the control state space is large or difficult to describe explicitly. These design problems include the specification of protocol controllers, communication devices, and computer interface subsystems. The high-level synthesis system described in this paper addresses these types of specification problems. This synthesis system is based on the Production-Based Specification (PBS) [29] [30] [31].

In a Production-Based Specification, the control structure of the design is specified as a hierarchical set of productions. Each production is viewed as a sub-machine or, more precisely, a non-deterministic automaton. Productions are recursively defined through hierarchical compositions of other productions. The hierarchical composition defines the control structure of a design *implicitly.* Data-flow computations called “actions” are hooked into this implicitly described control-flow by associating them with productions. A data-flow action is “executed” when its associated sub-machine is “recognized”. The recognition of a production may span many levels of abstraction. For example, the recognition of a production may correspond to the occurrence of a single signal transition, or to the termination of an entire protocol transaction.

Clairvoyant is a new high-level synthesis system intended for two areas of design. These areas are the specification and synthesis of designs:

1. that are naturally specified with the use of a grammar-based decomposition of the design’s behavior. These machines include those that perform computations in response to complex communication protocols.
2. that are naturally described as hierarchical compositions of interacting sub-machines. These machines include complex data-path controllers.

This manner of specification is intended to be a hardware analogy of the popular software engineering techniques and tools such as those used to create parsers, compiler control structures, and lexical analyzers, applied to high-level synthesis. Consider the design of an ASIC interface to an Ethernet. The sequential structure of the Ethernet protocol can be described using a set of productions in the Backus-Naur Form (BNF) commonly used for specifying language grammars [1] [14]. These productions define the syntax for correct Ethernet transactions as well as those transactions performed on the machine interface side of the interface. Every

possible combination of machine behavior on all interfaces is implicitly described this way as the set of recognizable sequences of the productions. It is then natural to attach data-path operations (actions) describing the desired semantics to this production framework since we assume that each action will be triggered on valid recognition of the underlying annotated production. For example, in the Ethernet interface, the action of storing a received data byte is triggered by the recognition of the production describing required sequence for a valid serial byte. This direct association between actions and the recognition of valid high level behavior allows for specifications where the required actions for a given behavior are described locally, but other possibly simultaneous actions necessary for other behaviors are described elsewhere. This property and the reusable hierarchy of productions provide the means for very concise and simpler behavioral specifications of these complex machines. Of particular utility is the ability to specify the desired behavioral response of a machine to a set of sequential stimuli without specifying a particular state machine implementation.

The Clairvoyant system is targeted toward the design of sequential machine controllers with associated data-paths for use in ASIC designs, where the constructed control structures aim for high performance and/or low power characteristics. These ASIC designs are typically multi-level logic circuits implemented using gate array or FPGA technology. Mapping the output into such implementations can be performed by any number of commercial synthesis packages, for example [7], because the output of Clairvoyant is a directly synthesizable subset of the VHDL [15] high level description (HDL) language. Thus, Clairvoyant works as an HDL generator. The user describes the behavior of a design entity in the form of a PBS description. This description is compiled and a hardware architecture is synthesized. In this process, both the control structure and the data-path register transfers required to implement the actions are created. The output is an HDL description of the architecture at the register-transfer level (sequential VHDL processes) with the required control machine described as a sequential logic network (structural VHDL). The Clairvoyant system aims to handle large designs with large state spaces. The control machine is output in a structural format to avoid the possibility that a deterministic state table output would require exponential space. This possibility arises from the non-deterministic nature of the input specification [14] [20]. The constructed number of flip-flops required in Clairvoyant's output structure is linearly related to the size of the input regular expression.

The designs specified using Production-Based Specifications are entities that are typically components of a larger system. The PBS specified design entity is assumed to interact with other design entities described at different levels of abstraction and using different specification techniques. In this way, the HDL generation can be applied to large constructions in exactly those places where it is most useful; i.e. sub-machines responding to complex sequential protocols or sub-machines connected to several other concurrently communicating sequential machines. An added advantage of the VHDL output is the ability of the designer to simulate and verify the synthesized design in the same way that conventional VHDL designs are constructed.

The next section of this paper describes the model and form of the Production-Based Specification. Related work is discussed in Section 3. In Section 4, the symbolic construction algorithms of the Clairvoyant system are described. In Section 5 experimental results are presented. Conclusions and future work are presented in Section 6.

2.0 The Production-Based Specification

The Production-Based Specification describes the behavior of a single design entity with a well defined boundary and interface. It is assumed that the design entity contains synchronous logic and that at least one of the input signals is a global clock signal. The PBS specification assumes a mono-rate sampling paradigm that allows multiple clocks for multi-phase synchronous clocking. Each PBS specified entity can be specified over a unique synchronous domain. The global clock(s) are assumed to be shared with other design entities in the complete system. It is assumed that other entities in the system interact with the synthesized entity only through its interface. The implementation of the entity is not important so long as it meets the desired sequential constraints of the interface and design constraints of area, cycle time, and power consumption. Thus all implementations of the design entity which satisfy the PBS specification are behaviorally equivalent, and differ only when characterized by implementation costs or other design metrics.

A *production* is a named composition of symbols, operators, and action clauses. There are two types of productions, those specifying sequential behaviors and those specifying combinational Boolean functions. The symbols in a *sequential production* are either references to other sequential productions or they are *tokens*. A token is a reference to a *Boolean Production* or a

Boolean composition in a sequential production. The symbols in a Boolean production are either references to other Boolean productions or they are atomic symbols. These atomic symbols either represent the input interface signals (primary inputs) or they are other language defined symbols.

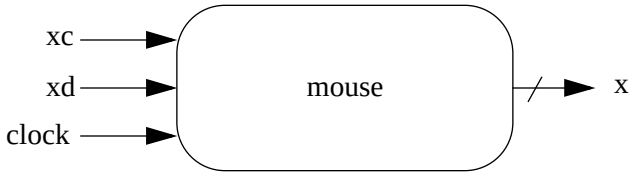
Composition operators are used to compose the productions. They are used to build more abstract or complex productions from simpler productions. The composition operators are similarly grouped into sequential and Boolean types for use in the two kinds of productions. Thus Boolean composition operators are used to define complex Boolean functions from simpler Boolean functions and sequential composition operators define abstract sequential behaviors from more primitive sequential behaviors. Table 1 describes the available composition operators.

Table 1: Composition Operators

op	name	meaning
a , b	concatenation	Accepted if a is accepted followed by the acceptance of b
a^n	multiple concatenation	n concatenations of the sub-machine a. n is an integer.
a || b	sequential or	Accept if a *or* b *or* both are accepted.
a&&b	sequential and	Accepted if a *and* b are simultaneously accepted.
! a	sequential not	Accepted if a is not currently in the state of acceptance.
a !! b	exception handler	a is initiated, if a will enter a state from which it can *never* accept, then b is initiated.
a !R	exception reset	a if initiated and re-initiated if a will enter a state from which it can *never* accept.
a*	Kleene Closure	Recognizes all sequences consisting of zero or more concatenations of a.
a+	one-or-more	Recognizes sequences of one or more a's. Equivalent to a*, a.
a | b	Boolean or	Boolean function $a \vee b$.
a&b	Boolean and	Boolean function $a \wedge b$.
~a	Boolean not	Boolean function $\neg a$.
a : b	qualification	Modify b such that for b to be accepted, a must be true throughout the execution of b.

A token is "recognized" or "accepted", if its Boolean function is satisfied in the context (clock cycle) in which the token appears in the productions. A production is accepted during the clock cycle in which the time sequence behavior dictated by its *composition* is satisfied. Thus token recognition provides the mechanism for the machine's sequencing behavior. The productions are annotated with action clauses or *actions* for short. An action is a specified data-flow computation that is executed when its antecedent symbol, composition, or production is recognized. In general, any number of productions may be active or simultaneously in the state of acceptance. A production may also accept several times in its execution.

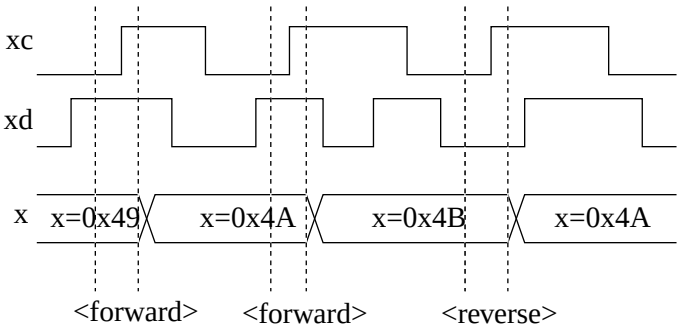
Recursive productions in the PBS specification are illegal since the intent is the specification of state machine controllers and data-paths of finite size. Although some recursive production sets can be constructed as FSMs, these cases are not currently allowed to simplify the implementation. This does not restrict the language capability since tail recursive behavior can be concisely described using the Kleene closure operator. A production, however, may be referenced by any number of *other* productions.



a) Design Entity

```
mouse   -> .*, event;
event   -> forward || reverse;
forward -> high:rising;  { x <= x + 1; }
reverse -> low:rising;   { x <= x - 1; }
rising  -> (~xc)+, xc;
high    -> xd;
low     -> ~xd;
```

b) PBS Specification



c) Timing Diagram

Figure 1. Mouse Example

Figure 1, illustrates an example design entity. The top portion (Figure 1a) depicts the design entity and its signal interface. The PBS specification for this design's behavior is shown in Figure 1b. In the description, there are seven productions: `mouse`, `event`, `forward`, `reverse`, `rising`, `high`, and `low`. Of these, the first five are sequential productions, and the last two are Boolean productions. The Boolean composition `(~xc)` is a token as it appears in the `rising` production. The symbols `xc` and `xd` refer to the input interface signals. By default, the first production in the PBS (`mouse`) is the *top-level production*. The top-level production encompasses the behavior of the whole design entity.

This description specifies the behavior of a 1-d positioning machine such as that used in a computer mouse pointing device. It continuously updates the signal `x` with a current 1-dimensional position based on the quadrature encoding of the signals `xc` and `xd` received from external motion sensors. Updating the position occurs if one of the productions `forward` or `reverse` are recognized. The `rising` production recognizes a rising edge occurring on the signal `xc`. It is recognized if `xc` is in a high state following a sequence of one or more cycles in which the signal `xc` has remained low since the initiation of `rising`. The productions `forward` and `reverse` are defined as qualified versions of the `rising` sub-machine. HDL action clauses are attached to these two productions. For example, if `forward` is recognized the signal `x` is incremented. The sequential composition "`.*,event`" represents the behavioral idiom "any input sequence followed by `event`" since "`.`" denotes the Boolean function that is always true. Thus, a new recognition of `event` is attempted on each clock cycle so that both the `forward` and `reverse` sub-machines are concurrently enabled to recognize motion of the mouse. Figure1c shows an example time sequence behavior of the mouse design.

Production-Based Specifications are convenient to the designer since it is often possible to implicitly specify very complex sequential constraints in a concise format. Additionally, this specification is local in the sense that additional desired behaviors can be specified by adding additional concurrent productions. For example, the *rate* of mouse motion can be measured by adding a production which counts idle clock cycles and adding two more actions to the `forward` and `reverse` productions. In general, such changes to FSM state descriptions require global modification of the entire design. Another valuable property is the ability to reuse previously defined productions representing key activities without regard to the possible concurrency of their execution. For example, a `read` production defined for a bus protocol can be used in the definition of all desired bus activities even if those activities might occur concurrently. A similar description as a deterministic FSM would potentially require the cartesian multiplication of all of the possibly concurrent `read` sub-machines to describe the possible states.

The PBS language was designed to allow flexible specification of finite state machine controllers. Although it contains a superset of the regular expression operators, the language remains in the class of finite automata. This is because all finite PBS specifications imply finite controllers. The controller does not require unbounded storage as is the case, for example, in a LALR parser which requires a stack [1] [14]. The extended operator set allows for more convenient expression of behaviors that would require exponentially larger specification in the form of traditional regular expressions, however each specification remains finite with respect to the controller.

2.1 Execution Model

The behavior of the PBS design entity is defined by execution of the implied control-flow where the actions are executed at their respective points in the protocol, and the execution of each action concludes on the accepting clock cycle. This model describes the external behavior of the design, not how the design is implemented. The PBS language model assumes sufficient resources to execute all potential simultaneous actions over all possible input excitation sequences. There may be considerable freedom to schedule the actions without violating the sequential interface constraints and thus optimize the resources or other design constraints. The PBS language doesn't preclude action execution overlapped with recognition of productions, so long as these transformations result in equivalent behavior. Techniques to exploit this freedom is the topic of current research and will be described in a separate publication.

Because of the nature of production recognition, it is possible that several actions may be triggered simultaneously (during the same clock cycle). Since such actions may have data dependencies, the conceptual ordering of their execution within the accepting cycle is important. Consider the actions: `{x := 0;}` and `{x := x + 1;}`. In one ordering, the resulting value of `x` is 0, while in the other order `x` is 1. *Action precedence* is defined for two actions if one of their respective productions is in the execution *scope* of the other respective production. The *scope* of a production includes all of the more primitive productions with which it is defined. The precedence ordering specifies that actions of more primitive productions conceptually occur

before those of less primitive productions. Thus, the set of actions has a partial order imposed by the production hierarchy. Actions whose productions have unrelated scopes do not have a defined precedence ordering.

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, since the recognition of the `block` occurs synchronously with the recognition of the last `word`. The action precedence rules imply that in the acceptance of the `block` production, the net result is that `x` is 0, since the reset action is conceptually last. The designer can exploit action precedence by crafting actions that supplant the results of others. When no action precedence is defined, dependencies between actions can be ambiguous. The synthesis system, however, can warn the user of a possible action conflict. This behavior is not forbidden so as to not limit the expressability of the language.

Synthesis of the controller in the Clairvoyant system does not rely on predicting the external world's response to the execution of an action. Thus, actions that "side-effect" via feedback from the external world and through the primary inputs, by design or otherwise, don't present synthesis problems. For example, an action may assert a signal on an output that is fed back to a primary input, thus changing how tokens are interpreted in subsequent cycles. These effects are considered in the construction because, effectively, every possible input sequence is assumed possible.

2.2 Operators

The sequential operators are a superset of the classical regular expression operators [1] [14] [20]. These operators include generalizations such as the *sequential not* "!" operator and the *sequential and* operator "&&" useful for specifying synchronization.

The exception operators are designed for specifying exception handling, re-synchronization, and recovery mechanisms. These operators are used to specify behaviors based on the conditions in which a sub-machine enters a state in which it can *never* accept. They are used to construct productions which recognize when a dependent production cannot accept and then take appropriate action. Exception operators may be nested hierarchically, as they operate on a general sub-machine which could contain other exception operators. An exception operator is defined over the production scope of all more primitive productions used to construct its dependent part. For example, consider the following nested productions:

```
a -> b!!c;
b -> d!!e;
d -> …
```

If the production `d` receives an input for which it has no more possible accepting sequences it is said to have failed. The `b` production can then be accepted only if the `e` production (the exception handler) is accepted. If `e` succeeds, `b` is accepted and so `a` is accepted as well. If `e` fails, then `b` fails and since `b` is in the scope of `a`, the exception handler `c` is activated. This type of behavior greatly simplifies the problem of specifying exceptional behavior since the alternative would be to specify every possible failure sequence for a production. This could require an exponentially larger regular expression.

The Boolean operators *and* "&", *or* "|", and *not* "~", are used in Boolean compositions for the specification of Boolean functions which are used as tokens in sequential productions and used as the left-hand operand in the qualification operator. A sequential production or composition may be *qualified* with a Boolean production or composition using the *qualification* operator. For a qualified production to accept, the Boolean part must remain true during any accepting sequence for the sequential part. In other words, the behavior of the qualified sub-machine is the same as the unqualified sub-machine in which all of its tokens have been *and*ed with the qualifying Boolean 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 referenced from several other productions in different contexts and its behavior refined through qualification in each instance.

3.0 Related Work

Jackson [16] championed a methodology for specification and design of software programs and software interfacing between programs, using constructive methods. Similar ideas are manifest in the successful compiler construction tools such as YACC [17] and LEX [21]. In these tools, the specification of the language to be compiled is described as a set of productions representing the language grammar. The semantic actions performed by the compiler are specified as code annotations to the

grammar. The tools compile this specification into the control structure of a compiler program to parse the specified language. This provides an enormous simplification in the complexity of constructing compilers since the designer need not consider the all the concurrent combinations of productions which are possible when the compiler is executed. PBS mimics this specification approach, however, fundamentally different operators and construction techniques are applied since the constraints differ between hardware and software. PBS achieves the great economy of specification characteristic of these tools.

Ullman et al. [10] [18] [35] studied the use and compilation of regular expressions in the design of hardware controllers. In this work, the design is specified as a single regular expression which is then implemented as a non-deterministic PLA. The non-determinism was expressed as feedback terms in the PLA, each of which indicating the validity of a given non-deterministic state. The system chose encodings based on a algorithm to minimize the number of feedback terms in the PLA. The PLA was minimized to produce the smallest number of cubes in the final design. Trickey [34] proposed a dynamic programming algorithm for optimizing the layout of these PLA pattern recognizers.

Although there are similarities between Ullman's approach and PBS, particularly in the use of regular expression operators, there are several differences in the specification form, construction techniques and goals. In PBS, the notion of explicit productions which are re-used is central and allows more concise specification. The modeling of actions in PBS is that of arbitrary high level data-flow behaviors instead of output signal transitions, and PBS implicitly allows multi-level logic models for the control.

STATEMATE [13] is a system for the design and documentation of reactive systems for use in interactive software and embedded systems. Designs are specified in the form of a hierarchical state chart [12]. In this specification, a state is active if *any* of its child states are active, for example, or alternatively if *all* of its child states are active. Transitions between states occur on events and are allowed between states at different levels of abstraction. The state charts are converted directly into software code. SpecCharts [27] addresses the behavioral specification of whole systems by combining hierarchical state charts and VHDL in a graphical specification methodology. The SADE system [23] uses graphical entry and underlying petri net models for design specifications that are converted into HDL code. The PUBSS system [38] specifies designs in the form of several interacting, but not hierarchical, cooperative VHDL processes that are modeled as behavioral finite state machines. Its synthesis focuses on scheduling the communication and computation in the design under the ensemble constraints induced from the individual process constraints. In all of these techniques, the designer *either* describes the behavior in terms of *explicit states* or the designer explicitly partitions the problem into interacting procedural processes that contain *explicitly defined state variables*. PBS, instead, describes the decomposition of the control behavior as a hierarchical listing of the possibly concurrent desired behaviors. This difference is similar to differences between C [19] and PROLOG [8] programs.

ESTEREL is a reactive programming language from which hardware specification has been recently studied [3] [11]. ESTEREL includes language constructs for parallelism and includes a powerful trap mechanism. There are several differences between PBS specification and ESTEREL. The primary difference is that the ESTEREL language is an *imperative* style language [11] and PBS is an *applicative* language for control specification. In an applicative language, the basic statements are definitions as opposed to assignments or sequences of tasks. Another difference is that a PBS specification has an explicit partitioning of the behavioral specification between the productions and the actions. Productions represent the implicit control behaviors the designer wants to specify at a very high level without describing the detailed state transitions or the linking of the control and data-paths. The ensemble of actions implicitly describe the data-path requirements. Thus, control of the data-path is implicit in PBS which simplifies the specification and allows more freedom in the final design implementation. ESTEREL's trapping mechanism is different from PBS's exception mechanism, however, both allow for the description of exceptional behaviors and both mechanisms use the notion of lexical scope.

In our previous work [29], we proposed the use of Production-Based Specification for use in high-level synthesis. In this earlier system, the execution model of the implied non-deterministic machine includes action clauses of VHDL code. These concepts have been greatly expanded in the synthesis system described in this paper. Clairvoyant incorporates fundamental improvements to the expressive power of the production specification language as well as more powerful synthesis techniques. The addition of new production composition operators, combinational Boolean productions, and the incorporation of recent BDD and symbolic representation techniques allows an efficient re-formulation of the synthesis task.

4.0 Clairvoyant Implementation

4.1 Design Representation

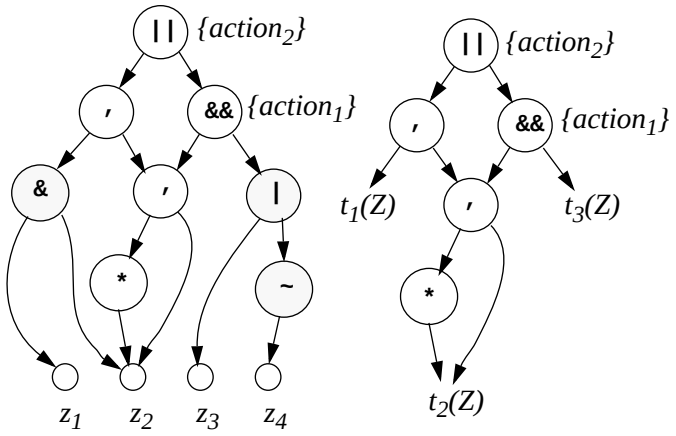
The synthesis process begins with the parsing of the Production-Based Specification. A *production representation* is created which captures the hierarchical structure of the description and it is derived from the production parse trees [1]. This representation is the starting point for further synthesis tasks and is retained throughout the synthesis process as an important representation of the high-level design structure. To describe the production representation and subsequent construction we will use a small example shown in Figure 2.

```
p1 -> p2 || p3; { action2 }
p2 ->(z1 & z2), p4
p3 -> p4 && p5; { action1 }
p4 -> z2+;
p5 -> z3 | ~z4;
```

Figure 2. Small Example

The symbols, `z1`, `z2`, `z3`, and `z4` represent external interface signals. The productions `p1`, `p2`, `p3`, `p4` are sequential productions while production `p5` is a Boolean production. Tokens are atomic sequential productions and represent sampling the interface signals for the desired Boolean function on a synchronous clock. For example, the composition "`(z1 & z2)`" is a token which is recognized if both `z1` and `z2` are true during the sample period. Production `p3` is recognized (and $action_1$ is triggered) if productions `p4` and `p5` are simultaneously recognized. i.e. `z2` became true at least one clock before the current clock and `z3` is true while `z4` is false. Production `p2` first requires `z1` and `z2` to be simultaneously valid and then `p4` to be recognized, while `p1` is recognized if either `p2` or `p3` are recognized. Note that if `p3` is recognized, both $action_1$ and $action_2$ will be triggered simultaneously, with $action_2$ conceptually occurring after $action_1$ due to action precedence (this precedence ordering can be easily seen for this example in Figure 3).

A collapsed production structure called the *production DAG* is subsequently constructed from the parse tree. Each node in the DAG represents a sequential composition operator. It is constructed from the production representation by propagating all Boolean operators toward the leaves of the DAG and then representing the resulting complex Boolean composition dags as combinational Boolean functions. For example, $t_1(Z) = z_1z_2$, and $t_2(Z) = z_2$. This construction is always possible because it is illegal for sequential productions or compositions to be used in Boolean compositions. References to these Boolean functions from the sequential composition nodes imply token recognition if the function is valid during that clock cycle. Practically speaking, these functions are represented by an ROBDD [5] [6] using the external signals as the basis variables. The example production representation and the collapsed production DAG are illustrated in Figure 3a and Figure 3b, respectively. Here, the sequential composition



a) Production Tree **b) Production-DAG**

Figure 3.

operator nodes are represented by unshaded nodes, while the Boolean composition nodes are shaded. It is important to note that each sub-DAG from a sequential node to its leaves represents a viable sub-machine of the design. Thus, the production DAG represents a hierarchical finite-state machine partitioning of the entire design. This property is exploited by the deterministic machine construction process detailed in the following sections.

4.2 Intermediate Machine Representation

The production DAG represents the input behavioral specification of the desired state machine. Construction of a physical implementation from this description passes through an intermediate stage in which state encodings have been made and the control can be described as a set of combinational functions taking the current state and inputs into the next state and outputs. This level description is output as register-transfer level VHDL for later logic synthesis and optimization by conventional tools. The internal design representation of this level is called an *intermediate machine*. The construction of this representation by conventional algorithms is hampered by the possibly exponential growth of the state transition table due to the inherent parallelism of the input specification. For this reason, an implicit construction

technique was devised allowing more flexible and larger problem instances than are tractable conventionally.

The intermediate machine representation consists of two parts, a state transition function and an output function for the machine. In what follows, B represents the set {0, 1}. The transition function Δ is a function mapping: $B^n \times B^k \rightarrow B^n$. This mapping is written:

Δ: $\{(x_1, x_2, x_3, \ldots, x_n)\} \times \{(z_1, z_2, z_3, \ldots, z_k)\} \rightarrow$
$\{(y_1, y_2, y_3, \ldots, y_h)\}$

where X, Y, and Z are Boolean vectors. X represents the present state of the machine, Z represents the input interface signals, and Y the next state of the machine. The transition function Δ represents a deterministic state transition function. The representation, however, is unconventional in that each state bit is associated with token recognition of a leaf of the production DAG. In this encoding of state, a true bit implies that control has been transferred to this bit and that the corresponding token (Boolean function of the signals) was recognized. Since the machine is non-deterministic, it is possible for several such bits (called *control points*) to be simultaneously true. Looking ahead, Δ can be viewed as a circuit -- in the example in Figure 3, recognition of the function $t_1(Z)$ is associated with state bit x_2 in the circuit in Figure 5. This representation has two views: As a whole, Δ represents the transition function of a deterministic FSM, while each function $y_i = f_i(X, Z)$ in Δ represents the excitation of an individual non-deterministic control point.

The Moore output function $\Lambda : B^n \rightarrow B^m$ is defined as a mapping:

Λ: $\{(x_1, x_2, x_3, \ldots, x_n)\} \rightarrow \{(a_1, a_2, a_3, \ldots, a_m)\}$.

where X is the present state and $a_i \in A$ represent each of the individual actions. Each action is triggered by the condition $a_i = c_i(X)$ corresponding to its location in the production DAG. Because many actions may be triggered simultaneously, action precedence enforces the execution sequence. The ordering of the a_i's in the vector A satisfy the partial order action precedence relations implied by the production DAG.

Alternatively, a Mealy form output representation Λ' is derived from Λ. In this case Λ' maps $B^n \times B^k \rightarrow B^m$, with the individual action conditions a function of X and Z, e.g. $c_i(X, Z)$. The action execution in the Moore form of the output function lags by a cycle vs. the Mealy form of the output function. The choice between the two forms of output function is selected prior to the construction by the designer.

4.3 Intermediate Machine Construction

The construction is a recursive procedure on the production DAG building the intermediate machine. This procedure applies a particular construction rule at each composition node of the DAG, based on the node's type. These rules are templates for application of a sequence BDD operations. Each time a leaf of the production DAG is reached, a new control point is added to the intermediate machine state vector. Since the production DAG may have several paths to a leaf from production re-use, the number of control points may be larger than the number of leaves in the DAG. This can be seen in the example in which the $t_2(Z)$ leaf denotes 4 distinct control points x_3, x_4, x_5 and x_6 in Figure 5. These control points represent sequentially distinct recognitions of the $t_2(Z)$ Boolean function of the input signals. Unlike Thompson's construction [1] [14] [20], here there is no need for ε-transitions to link the machine components. This is a consequence of the symbolic (ROBDD) representation of the control points excitation function which allows direct manipulation by the construction rules for both the conventional and generalized regular expression operators.

The construction is performed by the recursive procedure Build() illustrated in Figure 4. At each level of the recursion, the routine is passed a pointer to a node of the production DAG and a Boolean function (BDD node pointer) representing an excitation function f(X) passed from other recursion levels. The routine returns a Boolean function h(X) which is true on recognition of the current sub-DAG. At leaf nodes, new control points are allocated and their excitation functions are determined. When a leaf node is traversed, if a prior allocated control point exists with identical excitation, this prior control point is used instead of allocating a new control point. This is implemented using a memory function and is illustrated by the SaveControlPoint() and RecallControlPoint() calls in the pseudocode. At intermediate nodes, left and right sub-machines are composed via operations on the passed and returned functions. The construction process is initiated by allocating an initial control point x1 and calling Build(n=top-level-node, f(X)=x1).

The time complexity of this algorithm depends on the representation used for Boolean functions. Although ROBDD representations can exhibit exponential growth in general, in this algorithm, the variable support of the excitation functions returned from the left and right sub-machines is disjoint in all cases other than the exception operator constructions. The BDD growth is additive

```
Input : production-DAG n, Context f(X)
Output : Context h(X)

Build (n, f(X)) {
   if (n is a terminal function,
t_j(Z)){
      g(X, Z) = and(f(X), t_j(Z));
      x_t = RecallControlPoint(g(X, Z));
      if (x_t is not null) {
         x_i = x_t;
      } else {
         x_i = new control point;
         SaveControlPoint(x_i, g(X, Z));
      }
      y_i = f_i(X, Z) = g(X, Z)
      h(X) = x_i;
   } else if (n is a "concatenation"
node){
      g(X) = Build(node->left, f(X));
      h(X) = Build(node->right, g(X));
   } else if (n is a "sequential and"
node){
      g(X) = Build(n->left, f(X));
      h(X) = Build(n->right, f(X));
      h(X) = and(g(X),h(X));
   } else if (n is a "sequential or"
node){
      g(X) = Build(n->left, f(X));
      h(X) = Build(n->right, f(X));
      h(X) = or(g(X),h(X));
   } else if (n is a "sequential not"
node){
      g(X) = Build(n->right, f(X));
      h(X) = not(g(X));
```

Figure 4. Build Algorithm.

under the variable ordering implied by the sequential allocation of control point variables for these cases. As well, each constructed excitation function typically has very small variable support. Thus, for a DAG representing a regular expression, the time complexity of this construction is typically linear in the size of the regular expression.

The construction for the closure operator case is somewhat subtle. A temporary variable x_{tmp} is allocated and used in lieu of f(X) for construction of the operand sub-machine. This is done because the complete excitation function for the sub-machine depends on the function g(X) returned from Build(), which is unknown until the operand sub-machine is constructed. After Build() returns with g(X), the function h(X) = f(X) + g(X) is calculated. At this point, this function is substituted for x_{tmp} in every function in which x_{tmp} appears in the structure of the sub-machine. These substitutions are nicely performed by composing BDD functions e.g. f(x=g()) = ite(g(), f_x, f_x) [5] [6]. Note, a unique x_{tmp} variable must be used for each simultaneously open closure in the construction process.

Special sequential operators called *exception operators* are implemented. In an exception construction, a handler machine $\mathcal{M}_h$ is initiated when its associated sub-machine $\mathcal{M}$, once initiated, will enter a state in the next cycle from which it can *never* accept. Note this is a different notion than the *sequential not* operation in which both the cases of "active but not presently accepting" and "will never accept" are recognized. The function e(X, Z) represents the excitation that triggers $\mathcal{M}_h$. Consider the following equation for e_x(X, Z), which is used to calculate e(X, Z):

$$e_x(X, Z) = \overline{g(X)} \cdot \prod_{f_i \in \mathcal{M}} \overline{f_i(X, Z)} \qquad \text{(EQ 1)}$$

This equation describes the conditions in which $\mathcal{M}$ is not in a state of recognition, $\overline{g(X)}$, and will contain no active control points in the next cycle since each excitation function is false. To calculate e(X) we also need knowledge that $\mathcal{M}$ is active. This information can be computed as summation of the present control points in $\mathcal{M}$ and $\mathcal{M}$'s excitation. Thus e(X, Z) can be calculated as:

$$e(X, Z) = \left(\sum_{x_i \in \mathcal{M}} x_i + f(X) \right) \cdot e_x(X, Z) \qquad \text{(EQ 2)}$$

An alternative calculation for e(X, Z) can be derived using an extra control point to denote that control was passed to $\mathcal{M}$. This reduces the necessary logic necessary but introduces control points that do not purely represent token recognition. To derive e(X, Z) in this case, let x_h represent this control point. Then,

$$e(X, Z) = (x_h + f(X)) \cdot e_x(X, Z) \qquad \text{(EQ 3)}$$

The excitation of x_h is f_h(X, Z) and can be computed as follows:

$$f_h(X, Z) = (x_h + f(X)) \cdot \overline{e_x(X, Z)} \cdot \overline{h(X)} \qquad \text{(EQ 4)}$$

These exception operator constructions are valid for a general sub-machine, including sub-machines containing exception operators, and thus implement the notion of exception scope described in Section 2.

The circuit illustrated in Figure 5. represents the constructed intermediate machine for the example in Figure 3. Note that x_2 becomes valid after the machine is initialized only if t_1(Z) is seen on the inputs in the next

cycle. The control points x_3 and x_5 correspond to repetitive recognitions of $t_2(Z)$ required by the closure operator.

4.4 Action Conditions

In the Build() algorithm, the action execution conditions $c_K(X)$'s are constructed using the current h(X) at production operator nodes with the respective associated actions. The Moore output function Λ is constructed in this process. The Mealy output function can be created from the Moore output function. This is done by substituting $f_i(X, Z)$ for all x_i's in $c_k(X)$ forming a new $c_k(X, Z)$ by composing BDD functions. The Moore and Mealy machines are not equivalent; actions are triggered a cycle earlier in the Mealy format machine than in the Moore machine. In the Clairvoyant system, the designer chooses between the two forms of the action conditions before writing the PBS specification.

The action execution conditions for the Moore and Mealy implementations of the example design are as follows:

- Moore:
  action $c_1(X) = x_7x_6$
  action $c_2(X) = x_7x_6 + x_4$
- Mealy:
  action $c_1(X, Z) = t_2t_3x_1(x_5 + x_1)$
  action $c_2(X, Z) = t_2t_3x_1(x_5 + x_1) + t_2(x_3 + x_2)$

4.5 Action Ordering and Resources

Actions are comprised of register transfer operations destined for execution on data-paths associated with the synthesized controller. Action precedence from the production DAG is used to constrain the conceptual ordering of these executions. However, the output HDL must be carefully structured to allow subsequent synthesis procedures to take full advantage of exclusive control paths in the design to minimize resource usage [36]. In conventional high-level synthesis, the exclusive nature of the different control paths are usually apparent from the input description HDL code. In Clairvoyant, however, the control structure can be analyzed to find which actions can execute simultaneously and thus cannot share resources. The output HDL is structured to indicate the exclusive use of the register transfers and to meet the constraints of the partial ordering relations from action precedence. Note, if actions are further broken into operations, detailed scheduling could be performed using data-flow precedence as well, however, discussions of detailed scheduling in this context is the topic of future research.

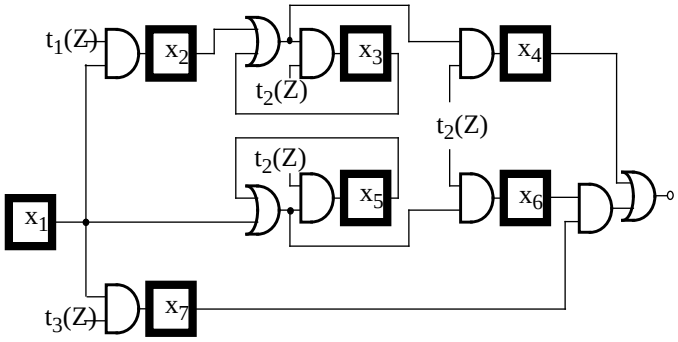


Figure 5. Example Circuit.

To determine if two actions can share resources, we need to determine if states exist in the machine in which both actions are simultaneously triggered. Since the action execution conditions are functions of the control points (state) and, in the Mealy case, the input interface signals, we can use the symbolic Boolean representation to determine if such states exist. Two actions a_i and a_j are mutually exclusive if the following equation holds:

$$c_i(X, Z)c_j(X, Z)R(X) = 0 \qquad \text{(EQ 5)}$$

In this equation, R(X) is a *characteristic function* [9] [22] [33] representing the set of possible deterministic states reachable from the initial state of the intermediate machine. This function mapping $B^n \rightarrow B$ is true if and only if the input vector $X \in B^n$ is a reachable state.

Assessing action conflicts between all pairs of actions is not sufficient, however, to determine the complete action conflict information. For example, consider three action conditions all executable on a common type of operator resource. If each pair of actions is used simultaneously in some state, but all three never occur together, only 2 data-path resources are needed even though no pair of actions are exclusive. This sharing cannot be predicted from a pair-wise analysis but is correctly handled in the Clairvoyant model which represents all action conflict information in a characteristic function A(Q). Q is a vector of variables $(q_1, q_2, ..., q_m)$ corresponding to the set of actions $(a_1, a_2, ..., a_m)$. A(Q) is true if there is a state in which the set of actions corresponding to true variables q_i occur simultaneously and thus can't be shared. A(Q) is computed as follows:

$$A(Q) = \exists(Z, X)(\prod_{i=1}^{m}(c_i(X, Z) \equiv q_i) \cdot R(X)) \qquad \text{(EQ 6)}$$

The existential quantification (smoothing) [9] [22] [33] operation above is defined as:

$$\exists(X) f = \exists(x_1)\exists(x_2)\ldots\exists(x_n) f \qquad \exists(x) f = f_x + f_{\bar{x}} \qquad \text{(EQ 7)}$$

The characteristic function A(Q) represents the image [9] [33] of the reachable state set R(X) projected onto the space B^m through the action condition functions.

To see how A(Q) can be used to construct the output control structure, consider A(Q) as a BDD. We can impose an order on the variables Q that minimizes the BDD size and that is compatible with the partial order required by the precedence relations. It is very likely that the actions naturally occur in independent sets which have no state overlap with other such sets. If the variables are ordered into such sets, the canonical nature of the ROBDD representation forces all the paths from the previous set into a unique node at the start variable of the next set. Then, since the BDD can be interpreted as a network of *if-then-else* constructs, we can construct a feasible control structure for the output using if statements and procedures which is no more complex than the BDD representation of A(Q) and correctly represents all possible resource sharing of the actions. This can be done in time proportional to the size of A(Q), even though the number of complete paths through the entire control structure may grow exponentially fast. Alternatively, A(Q) can be used to generate a a table of overlaps for pair-wise exclusion or other approximate analyses. Conflict analysis utilizing A(Q) is used to generate the output VHDL coded to maximize the effectiveness of subsequent high level synthesis allocation and resource sharing algorithms in processing the generated VHDL code.

4.6 Reachable State Analysis

Clairvoyant is equipped to perform a reachable state analysis on the constructed intermediate machine to compute the set of possible deterministic states reachable from the initial reset state of the intermediate machine: $x_1\bar{x}_2\bar{x}_3\bar{x}_4...\bar{x}_n$. *Reachable state analysis is not required for the synthesis of the intermediate machine*, but it is useful in several ways. In particular, knowledge of the reachable states is needed for the exact construction of A(Q) shown previously. Reachable state information can also be used to simplify portions of the intermediate machine, for example, simplifying (EQ 1). The essential use is to describe all deterministic states of the machine. All state bit combinations not in this set are not states and therefore specify don't care conditions for any of the functions depending on the control points.

The computation method is based on the recent implicit fixed-point iteration techniques [9] [22] [33] with custom heuristics based on properties of the intermediate machine. Even using these techniques, calculation of the set of reachable states is usually far more time consuming than the construction of the intermediate machine.

Recall that the set of reachable states is used in calculating the action relation. An approximate action conflict characteristic function can be calculated assuming all states are reachable in the event the reachable state computation is not invoked. For the Mealy model machines, this approximation is useful because particular actions are often strongly correlated to the current inputs. For example, in the mouse example described earlier, the increment and decrement actions are selected by the level of a single input, so they are clearly exclusive.

4.7 Machine Locality Property

A useful property of the intermediate machine representation is that any node of the production DAG can be directly related to specific portions of the intermediate machine representation, and each control point and excitation function can be related back to specific productions and compositions. Specifically, each production and each composition node is associated with a set of closed intervals [a, b] of control points created on each call to build() for the node. A new interval of control points is created each time the production is re-used since control points are allocated sequentially. This property is important for debugging, high-level optimization, and design information tracking. It can be used to provide links between the specification and structure similar to the CORAL II approach [4]. For example, the example productions in Figure 2 can be related to the circuit in Figure 5 as shown in Table 2.

Table 2: PBS ↔ Intermediate Machine Linkage

sequential production	control points	interval(s)
p1	{x2, x3, x4, x5, x6, x7}	[2,7]
p2	{x2, x3, x4}	[2,4]
p3	{x5, x7}	[5,7]
p4	{x3, x4, x5, x6}	[3,4] [5,6]

4.8 Implementation Details

Clairvoyant synthesis system was developed in C++ and is comprised of approximately 7600 total lines. Of this, 3160 lines represents reusable classes including a 1485 line BDD manipulation package. The output of the Clairvoyant PBS compiler is VHDL code describing the synthesized machine architecture. This VHDL is composed of structural elements that describe the logic structure of the controller, and processes that implement the register transfers and data-path logic required by the

actions. The structure of the VHDL action processes satisfy the partial ordering required by action precedence.

The tool uses BDDs for the symbolic Boolean manipulations. During the synthesis, BDD variables are allocated dynamically as the machine construction proceeds. This construction process also naturally creates a reasonable heuristic variable ordering based on circuit topology arguments [25]. BDD variables are grouped into classes based on use and are interleaved. The following three-way ordering is used: $z_1 < x_1 < y_1 < z_2 < x_2 < y_2 < z_3 < x_3 < y_3$ ... The y_i's represent an additional set of state variables used by the reachable state analysis, and in computing the action conflict relation A(Q).

In the Clairvoyant system, after the intermediate machine is constructed, redundant registers may exist. These arise for several reasons. Boundary registers with lack of fan-out may exist if action conditions are converted from Moore to Mealy form. Registers with identical excitation may exist that were not filtered by the memory function described in Section 4.3. This is due to the existence of temporary variables allocated in the construction process preventing identification. Finally, if the reachable state analysis is invoked, additional redundant registers may be identified using techniques similar to those described in [28]. Post-processing steps manipulate the intermediate machine to ensure that all registers (control points) identified as reductant will be eliminated by later logic synthesis. For example, after logic synthesis, the registers x_4, and x_6 will be removed (equivalent fan-in to x_3, x_5). If Mealy action conditions are used, register x_7 (output unused) will be removed as well, in the circuit in Figure 5.

5.0 Experimental Results

5.1 Examples

Several example designs were specified using Production-Based Specifications. These designs and their characteristics are tabulated in Table 3. The number of inputs includes the clock signal and the reset signal. Each design was verified by simulation of the VHDL output from Clairvoyant. The several mouse designs are different versions of the 1-D quadrature decoder machine described in the introduction of this paper. The "mouse(a)" design is identical to this earlier example. The "mouse(b)" design recognizes a complete quadrature sequence as an event and so is a more restrictive version although both versions correctly interpret quadrature data. The "xymouse" designs are 2-dimensional versions of the respective 1-D mouse decoder examples. The xymouse designs are specified as a single set of productions using the expressive power of the Boolean representation in the language. Using the early version of the PBS language [29], the xymouse designs would require a symbolic alphabet consisting of the cartesian product of the 1-D mouse alphabets, and would be far more difficult to express. Using arbitrary Boolean functions as tokens allows representation of enormous symbolic alphabets, and makes specification of realistic designs possible.

Table 3: Design Characteristics

design	productions	actions	inputs	outputs
mouse(a)	4	2	4	8
xymouse(a)	7	4	6	16
mouse(b)	8	2	4	8
xymouse(b)	15	4	6	16
count0	6	3	3	4
qr42	4	3	4	2
i8251ar	16	4	8	10
midi	30	12	3	16
mismatch	7	1	4	1

The **"**count0**"** example is a design that counts sequential zero's in a valid input frame format. This example is based on the procedural VHDL design in [7]. The "qr42" design is a handshake conversion protocol. This design is a standard asynchronous example specified in a PBS as a synchronous machine. This design connects two interfaces together, one side operating with two-phase (non-return-to-zero) signaling and the other with four-phase (return-to-zero) signaling. This machine uses of the "&&" operator for synchronization. The "i8251ar**"** example is the asynchronous receiver protocol in the i8251 high-level synthesis benchmark [2]. This example uses the Boolean qualification operators in the specification of the different modes of operation. This design also uses an exception operator to reset the machine if invalid stimulus is encountered. The "midi" design is a large design example. It is an interface controller which interprets the MIDI [26] music protocol for a digital synthesizer chip controller. The specification of this design also includes an exception operator to restart the machine in case of invalid input sequences. The "mismatch" example is the pathological regular expression described in [18] which detects mismatches between first and the last symbols in the input sequence. This example is expected to produce very large numbers of deterministic states.

5.2 Results

Results for compiling the example designs to the intermediate machine form are illustrated in Table 4. In this table, the number of control points in the

Table 4: Intermediate Machine Synthesis

design	control points	build time	ite calls
mouse(a)	8	0.08	306
xymouse(a)	15	0.12	1,047
mouse(b)	14	0.08	688
xymouse(b)	26	0.17	2,136
count0	7	0.10	1,004
qr42	21	0.19	3,421
i8251ar	14	0.34	10,004
midi	182	4.09	112,545
mismatch	69	0.28	7,465

intermediate machine representation after construction are listed. Also listed in Table 4 are the construction times in CPU seconds (SPARC® compatible Solbourne® Series 5e/906 machine) and construction complexity measured in terms of the numbers of calls to the primitive BDD function ite() for the entire construction.

Table 5: Reachable State Analysis

design	states	depth	sec	ite calls
mouse(a)	8	2	0.23	8,761
xymouse(a)	50	2	1.52	64,396
mouse(b)	14	4	0.71	28,863
xymouse(b)	170	4	10.72	408,505
count0	5	3	0.17	5,663
qr42	62	12	3.49	126,158
i8251ar	17	12	3.17	114,765
midi	166	40	1,791	37,331,185
mismatch	8062	16	5,191	172,797,476

Table 5 shows the results of the reachable state analysis. The number of reachable states represent the total number of unique deterministic states in the intermediate machine representation of the controller. The diameter measures the shortest path from the initial state of the controller to the furthest reachable state. This number is directly related to the number of fixed point iterations to compute the reachable states. The ite call numbers reflect the total number of calls to ite() during the reachable state analysis. Times are CPU seconds (Solbourne Series 5e/906 machine).

Action conflict data is given in Table 6. In this table, “conflict states” refers to the number of points in the Boolean space B^m covered by A(Q) in each of the designs. This represents the number of combinations of possible simultaneous action execution. For example, in the mouse designs three states are possible for its two actions. Neither action can execute or each action can execute individually, however, both can never execute simultaneously. The table also indicates the number of BDD nodes in the function A(Q) and the time (CPU seconds for Solbourne 5e/906) and number ite calls recorded to construct A(Q).

Table 6: Action Conflict Data

design	actions	conflicts	A(Q) nodes	sec	ite calls
mouse(a)	2	3	2	0.04	1,075
xymouse(a)	4	9	4	0.35	7,531
mouse(b)	2	3	2	0.06	1,701
xymouse(b)	4	9	4	0.9	18,869
count0	3	3	4	0.03	825
qr42	3	3	2	0.39	9,203
i8251ar	4	6	5	0.38	8,885
midi	12	11	26	49.0	676,379
mismatch	1	2	0	2.21	53,087

Table 7: Variable Support of Intermediate Machine Functions

	Variable Support			
	Δ		Λ	
design	avg.	max.	avg.	max.
mouse(a)	2	4	4	4
xymouse(a)	2	4	4	4
mouse(b)	3	4	4	4
xymouse(b)	3	4	4	4
count0	3	5	4	5
qr42	3	6	11	18
i8251ar	5	20	7	12
midi	3	167	11	27
mismatch	2	4	10	10

The intermediate machine is used in Clairvoyant for representation, analysis, and optimization of the design. It is also utilized in derivation of a circuit realization of the design’s controller. This is advantageous because the construction naturally creates machine implementations with very small excitation functions. In practice, the transition function to calculate a given control point tends to depend on a very small number of other control points. Results showing the size of the average and maximum literal support for the control points is tabulated in Table 7. This table reflects the variable support of the control point excitation functions (the $f_i(X, Z)$’s in Δ) and the action conditions (the $c_i(X, Z)$’s in Λ) after redundant registers are removed. Average and maximum numbers are reported in the table. The relatively large maximum support for the i8251ar and midi example is a consequence of the exception operators in these designs.

Comparisons of the encodings present in the Clairvoyant implementations of the example designs to conventional state assignment techniques are presented in Tables 8a and 8b.These comparisons were performed as follows. BLIF files describing the controller portion of the designs were generated from the intermediate machine representation by the Clairvoyant system. These BLIF files were read into the SIS sequential and logic synthesis system [32] for analysis. Comparisons were made between the SIS circuit network optimizations of the Clairvoyant implementations and those generated by extracting the State Transition Graphs (STGs) and performing state assignment. Three state assignment algorithms were used in the comparisons: NOVA [37], JEDI [22], and one hot. These algorithms were invoked from within SIS. Table 8a shows the comparison of the Clairvoyant encodings to state assignments of the extracted STG. Table 8b shows the same comparisons, however, the extracted STGs were state minimized before state assignment. In these comparisons, standard SIS minimization scripts were invoked for the network optimization..

Table 8a: Encoding Comparison #1

design	Clairvoyant			Extract STG, State Assign							
				state	nova			jedi		one hot	
	L	D	R		L	D	R	L	D	L	D
mouse[a]	18	5	4	5	21	7	3	19	8	26	4
xymouse[a]	36	5	8	25	100	29	5	99	21	168	14
mouse[b]	36	5	10	11	42	19	4	35	9	61	10
xymouse[b]	72	5	20	121	1948	33	7	1844	67	943	18
count0	16	5	4	5	12	8	3	16	6	16	6
qr42	77	6	21	62	359	68	6	317	54	382	19
i8251ar	84	16	14	17	74	16	5	112	25	95	12
midi	604	22	166	166	743	81	8	1098	89	705	23
mismatch	114	6	62	unable							

Table 8b: Encoding Comparison #2

design	Clairvoyant			Extract STG, State Minimize, State Assign							
				state	nova			jedi		one hot	
	L	D	R		L	D	R	L	D	L	D
i8251ar	84	16	14	14	71	24	4	75	14	95	11
midi	604	22	166	unable							
mismatch	114	6	62								

In the tables, “L” refers to the number of literals in the factored form of the optimized technology independent network. A measure for performance comparison of the encodings was obtained by mapping the optimized network to two input logic gates and recording the maximum levels of logic required. These numbers are listed in the columns labeled “D”. The number of required registers for each of the encodings is also listed in the table in columns labeled “R”. The STG for the mismatch example could not be extracted due to the large number of deterministic states. State minimization for the midi STG failed due to the example’s size. Note, in the extraction of the STGs from the networks, not all of the network reachable states are significant due to the presence of redundant registers which don’t fan out. This is why the number of STG states differs from the number reachable states in the intermediate form. The SIS command “xdc” reports the number reachable states of the network which are identical to those listed in Table 5.

Results for further VHDL and logic synthesis of the output RTL implementations generated by the system for each of the example designs is shown in Table 9. Gate level circuit implementations of the designs were synthesized using the Synopsys® VHDL and logic synthesis tools. In these results, no additional sequential optimizations such as state assignment, re-timing, or re-encoding were invoked. The logic synthesis was directed to optimize for speed (critical path delay) and the synthesized circuits were optimized for and mapped to LSI 10k gate array library cells [24]. The data for the path delay (in nS), relative area, total number of LSI 10k cells, and total number of flip flops is given. These numbers include both the control as well as the data-path portions of the designs. The relative area numbers are the area estimates based on LSI 10k library cells returned by the synthesis tool.

Some conclusions can be drawn from these results. In comparing the mouse machines with the xymouse machines, the number of productions and control points roughly doubles while the state space of the machine is

Table 8b: Encoding Comparison #2

design	Clairvoyant			Extract STG, State Minimize, State Assign							
				state	nova			jedi		one hot	
	L	D	R		L	D	R	L	D	L	D
mouse[a]	18	5	4	3	10	4	2	10	4	10	3
xymouse[a]	36	5	8	9	46	12	4	35	8	59	4
mouse[b]	36	5	10	7	29	10	3	26	16	41	8
xymouse[b]	72	5	20	49	528	65	6	515	50	636	22
count0	16	5	4	4	12	5	2	11	5	13	6
qr42	77	6	21	16	54	20	4	84	28	121	12

Table 9: VHDL and Logic Synthesis Results

design	delay	area	#cells	total #FF
mouse(a)	6.43	277	61	12
xymouse(a)	7.15	514	123	24
mouse(b)	5.98	324	83	18
xymouse(b)	6.56	601	159	36
count0	4.74	116	41	7
qr42	4.65	235	77	21
i8251ar	6.82	365	138	22
midi	13.96	1,927	532	194
mismatch	3.65	656	175	62

squared. It is clear that the machine construction complexity is not proportional to the growth of the machine's state space as would be expected from conventional algorithms. The execution speed of the two designs (which includes the data-path delay as well as the control delay) is nearly the same (the Clairvoyant design for the xy-version consists essentially of two of the single machines in parallel thus the delay differences are artifacts of the further synthesis). The midi design was much more complicated in its behavior and included an exception handling routine so that any valid data imbedded in arbitrary invalid data would be correctly interpreted. Considering this, the design's cycle time was an impressive 13.96 nS. Also, note that this design required only 30 productions for the entire specification, which fit easily on 2 pages of text. Finally, the pathological mismatch design had 8062 deterministic states, but was constructed in 0.28 CPU seconds, showing the relative independence of the construction time from the size of the deterministic state space. Table 6 shows that our optimal technique for generation of action exclusion information is both feasible and is simple to map into the output VHDL, as shown by the very small ROBDD representations representing the functions A(Q). Use of this information is critical in allowing subsequent logic optimization to minimize the required resources.

It is of interest to note the relatively high performance of the designs derived directly from the intermediate form. These designs typically have more registers than conventional designs but generally have very simple excitation logic between the control points. This is due to the direct use of the specification in constructing the logic and selecting the deterministic codes. In effect, the control points provide a set of signals from which the excitation functions can be derived with very small literal support. These considerations are demonstrated by the differences in logic complexity (as reflected by factored literal counts) and in controller logic delay (as reflected by the mapped logic depth) shown for Clairvoyant designs and designs created by symbolic state extraction, state assignment, minimization and identical synthesis. In particular, in the small state machines with little parallelism: mouse(a), mouse(b), i8251ar, and count0, the Clairvoyant designs are comparable to the state assigned designs. However, for larger and more parallel cases such as xymouse and qr42, the quality of the distributed encoding becomes much more impressive. Note that even when the minimal machine encodings have comparable literal counts, the logic depth (and hence the controller delay) of these machines is greater. In the typical case, the logic depth of Clairvoyant was smaller than any of the other encodings, state minimized or not. Finally, it is important to note that the mismatch design complexity is relatively simple even though it could not be synthesized at all using state-graph based techniques.

The register costs for the Clairvoyant designs must be measured relative to the implementation technology. The encodings are ideal for FPGA implementation where registers are virtually free since they typically occur in every FPGA cell. In these designs, the small average literal support and logic depth should allow efficient, high performance designs. In other technologies where high performance is required, these encodings may be desirable, regardless of the register costs.

6.0 Conclusions and Future Work

We have presented a new high-level synthesis system directed toward the synthesis of complex designs that are specified concisely using hierarchical grammar-like decomposition of their behavior. These specifications are of practical use in synthesis problems that are control dominated or require complex concurrent protocols. The use of productions enables the specification to span many levels of complexity, and to describe what actions should be taken in each case. Non-determinism in the language frees the designer from the onerous task of determining the precise behavior required of each deterministic state. Instead, the designer needs only to specify the kinds of behaviors expected and what actions should take place. The direct use of Boolean functions in both the token recognition and production qualification processes greatly expand the expressability of design specifications in this format. The resulting specifications are very concise and allow the designer to specify the design at high levels of abstraction in which the detailed interaction of the sub-machines is automatically derived. The system synthesizes a hardware architecture with VHDL register-transfer output allowing system assembly

with VHDL modules from many sources and use of commercially available tools.

The Clairvoyant system implementation makes extensive use of symbolic construction techniques to perform this synthesis. These techniques include a new direct machine construction algorithm which is not directly impacted by the size of the deterministic state space and hence is applicable to very large designs. The constructed intermediate machine form is a convenient representation base for further analysis and optimization using both classical and more recent symbolic techniques. With little additional optimization, this form yields sequential machines with superior performance characteristics. Techniques for evaluating resources conflicts for designs in this representation have also been described.

In future work, further optimization of the intermediate machine to reduce the number of registers without reducing the high level of performance achieved in the design will be studied and applied to the Clairvoyant synthesis tool. Additional studies and possible future work includes operation scheduling and optimization obeying the controller and protocol constraints as well as optimizations to simplify the productions.

Acknowledgments

The authors wish to acknowledge Emil Girczyc and Margaret Marek-Sadowska for helpful suggestions and discussion. The authors also thank the reviewers for their constructive feedback. This research was made possible through the generous support of Synopsys, Inc. and the University of California MICRO program — #92-019.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers Principles, Techniques, and Tools*, Reading: Addison-Wesley 1988.
2. *Benchmarks of the Fourth International Workshop on High-Level Synthesis*, 1989.
3. G. Berry, "A Hardware Implementation of Pure ESTEREL," Sādhanā, pp. 95-130, Vol. 17, Part 1, March 1992.
4. R. L. Blackburn, D. E. Thomas, and P. M. Koenig, "CORAL II: Linking Behavior and Structure in an IC Design System," proc. 25th DAC, pp. 529-535, June 1988.
5. K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient Implementation of a BDD Package," proc. 27th DAC, pp. 40-45, June 1990.
6. R. E. Bryant, "Graph Based Algorithms for Boolean Function Manipulation," IEEE Transactions on Computers, pp. 677-691, August 1986.
7. S. Carlson, *Introduction to HDL-Based Design Using VHDL*, Mountain View: Synopsys, 1990.
8. W. F. Clocksin and C. S. Mellish, *Programming in PROLOG*, Second Edition, Berlin: Springer-Verlag, 1984.
9. O. Coudert and J. C. Madre, "A Unified Framework for the Formal Verification of Sequential Circuits," proc. ICCAD-90, pp. 126-129, November 1990.
10. R. W. Floyd and J. D. Ullman, "The Compilation of Regular Expressions into Integrated Circuits," Journal of the ACM, pp. 603-622, Vol. 29, No. 3, July 1982.
11. N. Halbwachs, *Syncronous Programming of Reactive Systems*, Dordrecht: Kluwer, 1993.
12. D. Harel, "Statecharts: A Visual Approach to Complex Systems," Science of Computer Programming, vol. 8, pp. 231-274, 1987.
13. D. Harel, et. al., "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," proc. International Conference Software Engineering, pp. 396-406, 1988.
14. J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Reading: Addison Wesley 1986.
15. *IEEE Standard VHDL Language Reference Manual*. IEEE Std. 1076-1987.

16. M. A. Jackson, “Constructive Methods of Program Design,” *Lecture Notes in Computer Science*, Vol. 44, Springer-Verlag, pp. 236-262, 1976.

17. S. C. Johnson, “Yacc: Yet Another Compiler Compiler,” Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ 1975.

18. A. R. Karlin, H. W. Trickey, and J. D. Ullman, “Experience with a Regular Expression Compiler,” proc. ICCD, pp. 656-665, 1983.

19. B. W. Kernighan and D. M. Ritchie, *The C Programming Language.* Second Edition, Englewood Cliffs: Prentice Hall 1988.

20. Z. Kohavi, *Switching and Finite Automata*, New York: McGraw-Hill, 1978.

21. M. E. Lesk, “Lex -A Lexical Analyzer Generator,” Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, NJ 1975.

22. B. Lin, *Synthesis of VLSI Designs with Symbolic Techniques*, Ph.D. Thesis, University of California, Berkeley, UCB/ERL M91/105, November 1991.

23. J. Lathi, M. Sipola, and J. Kivelä, “SADE: A Graphical Tool for VHDL-based Systems Analysis,” proc. ICCAD-91, pp. 262-265, November 1991.

24. LSI Logic Corporation, *1.5-Micron Compacted Array Technology*, Databook, July 1987.

25. S. Malik, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincetelli, “Logic Verification Using Binary Decision Diagrams in a Logic Synthesis Environment,” proc. ICCAD-88, pp. 6-9, November 1988.

26. *MIDI Specification Version 1.0*, International MIDI Association, 1983.

27. S. Narayan, F. Vahid, and D. D. Gajski, “System Specification with the SpecCharts Language,” proc. ICCAD-91, pp. 266-269, November 1991.

28. H. Savoj, H. Touati, and R. K. Brayton. “Extracting Local Don’t Cares for Network Optimization,” proc. ICCAD-91, pp. 514-517, November 1991.

29. A. Seawright and F. Brewer, “Synthesis from Production-Based Specifications,” proc. 29th DAC, pp. 194-199, June 1992.

30. A. Seawright and F. Brewer, *PBS 2.x Users Guide*, ECE Tech. Report #92-21, UCSB, October 1992.

31. A. Seawright and F. Brewer, “High-level Symbolic Construction Techniques for High Performance Sequential Synthesis,” proc. of the 30th DAC, pp. 424-428, June 1993.

32. E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, “SIS: A System for Sequential Circuit Synthesis,” Electronics Research Laboratory Memorandum No. UCB/ERL M92/41, May 1992.

33. H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli, “Implicit State Enumeration of Finite State Machines using BDD’s,” proc. ICCAD-90 pp. 130-133, November 1990.

34. H. W. Tricky, “Good Layouts for Pattern Recognizers,” IEEE Transactions on Computers, pp. 514-520, Vol c-31, No. 6, June 1982.

35. J. D. Ullman, *Computational Aspects of VLSI*, Rockville: Computer Science Press, 1984.

36. K. Wakabayashi and H. Tanaka, “Global Scheduling Independent of Comtrol Dependencies Based on Condition Vectors,” proc. 29th DAC, pp. 112-115, June 1992.

37. T. Villa, T. and A. Sangiovanni-Vincentelli, “NOVA: State Assignment of Finte State Machines for Optimal Two-Level Logic Implementation,” IEEE Transactions on Computer-Aided Design, vol. 9. pp. 905-924. September 1990.

38. W. Wolf et. al., “The Princeton University Behavioral Synthesis System,” proc. 29th DAC, pp. 182-187, June 1992.