Non-Deterministic Specification for Sequential Synthesis

Forrest Brewer, Andrew Seawright, Andrew Crews
forrest@ece.ucsb.edu, andy@synopsys.com, crews@corona.ece.ucsb.edu

Department of Electrical and Computer Engineering
University of California, Santa Barbara
Why another sequential specification tool?

- **Deterministic Model encourages global state-based specification style**
  - Efficiency decreases rapidly with increasing numbers of states
  - Many properties of the machine make sense only for a subset of ‘bits’
  - Small changes in STG may lead to arbitrarily large changes in implementation

- **Designers are forced into partitioning the behavior into a set of manageable DFSM’s**
  - No guarantee that chosen partition is globally efficient
  - Very easy to create specification problems such as safety, deadlock, etc.
  - Can be difficult to obtain desired global behavior from local behaviors of submachines
  - Applicable tools force bottom-up design methodology
  - Engineering changes on global specification can be difficult to integrate in submachines

- **Current Tools support only very small DFSM’s**
  - State assignment optimized only for minimal (log), 1-hot or 2-hot, encodings
  - Tools limited by deterministic state complexity -- itself exponential in circuit size
  - Little correlation between STG and circuit implementation
  - Minimization of States/Latch # does not necessarily minimize design
Non-Deterministic Finite State Machines

- Multiple tokens (control points)
- Can describe non-deterministic behavior
- Any DFA graph is also an NDFA graph
- Any NDFA graph can be rewritten to become a DFA, if it is deterministic
- Circuit model is effectively an NDFA-- local states only depend on logic fan-in tree
- Any synchronous circuit can be modeled by a comparably linear size NDFA

- Use NDFA’s to describe DFA behavior in a more powerful representation

- Can use NDFA to succinctly describe external (sequential) behavior constraints (ref. Brayton and Watanabe)
An Example NDFA

- If a transition does not exist, the token is lost (destroyed)
- If multiple transitions for an input, tokens are created
- ‘1’ outputs dominate ‘0’ outputs

Sample Input Sequence

<table>
<thead>
<tr>
<th>input</th>
<th>state(s)</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>A,C</td>
<td>010</td>
</tr>
<tr>
<td>101</td>
<td>A,B,C</td>
<td>000</td>
</tr>
<tr>
<td>000</td>
<td>A,D</td>
<td>100</td>
</tr>
<tr>
<td>101</td>
<td>A,B,C</td>
<td>000</td>
</tr>
<tr>
<td>111</td>
<td>A,B,C</td>
<td>011</td>
</tr>
<tr>
<td>111</td>
<td>A,B,C</td>
<td>011</td>
</tr>
</tbody>
</table>
Eqivalence DFA Behavior

KISS Table

<table>
<thead>
<tr>
<th>i 3</th>
<th>o 3</th>
<th>p 32</th>
<th>s 6</th>
<th>r S2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-- S0 S2 100</td>
<td>0-- S3 S2 000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100 S0 S0 000</td>
<td>111 S3 S1 011</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101 S0 S1 000</td>
<td>110 S3 S0 001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110 S0 S0 001</td>
<td>100 S3 S0 000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111 S0 S1 001</td>
<td>101 S3 S1 010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01- S2 S2 000</td>
<td>0-- S4 S2 000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111 S2 S3 001</td>
<td>111 S4 S1 001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110 S2 S4 001</td>
<td>110 S4 S0 001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001 S2 S2 000</td>
<td>101 S4 S1 000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101 S2 S5 000</td>
<td>100 S4 S0 000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-00 S2 S2 000</td>
<td>01- S5 S2 000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0-- S1 S2 100</td>
<td>111 S5 S3 011</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100 S1 S0 000</td>
<td>110 S5 S4 001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101 S1 S1 010</td>
<td>001 S5 S2 000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110 S1 S0 001</td>
<td>101 S5 S3 010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111 S1 S1 011</td>
<td>-00 S5 S2 000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

State Transition Graph
Machine Specification via Event Sequences

- An alternative specification method is to describe those sequences of input stimuli (events) which cause desired outputs (actions)
  - a set of sequences defines DFA states implicitly
  - a timing diagram describes an event sequence, usually between 2 or more machines:

- above, m1 outputs ALE and ADD and then looks at WAIT to see when DATA will be valid
- m2 looks at ALE, latches the address, outputs WAIT until it is ready with the data

M1: [ALE][ADD], WAIT?, WAIT?, WAIT?, ~WAIT, Latch DATA
M2: ALE?[Latch ADD], [WAIT], [WAIT], [WAIT], [WAIT], [~WAIT], [DATA]
Machine Specification via Event Sequences II

- **Typically, a timing diagram represents an infinite set of such sequences**
  - no bound on the number of wait’s above...

- **A synchronous event sequence samples inputs and sets outputs on clock edges**
  - particular clocking scheme can be abstracted to “sampling” and “signaling” events
  - sampling and signalling events must be well ordered
  - logically dependent signals disambiguate timing via clock synchronization

- **A set of such sequences can be ‘open’ or ‘closed’**
  - a set is closed if any input sequence matches at least one event sequence
  - an implementation is necessarily closed, but not necessarily specified

- **A set of sequences can be “forbidden”**
  - a specification may contain sequences that are required, some that are don’t care, and some that must not occur
  - typically, there is design freedom implicit in how the don’t care sequences are implemented
Regular Expressions describe sets of event sequences

- RE’s can represent any finite automata, and many possible event sequences

• 3 Classical Regular Expression Operators
  - concatenation: “A,B” means “input A followed by input B”
  - closure: “A*” means “zero or more occurrences of A”
  - union: “A || B” means “either input A or input B”

• A 4th “operator”: the action.
  - Creates a “complete” language allowing arbitrary symbolic machine description
  - However, the description is not guaranteed to be concise or complete

| M1:  | (.+[ALE][ADD], WAIT*, ~WAIT, .[Latch DATA])* |
| M2:  | (ALE[Latch ADD], ~Dready*[WAIT], Dready[DATA])* |

• Above pair of Regular Expressions describe all possible sequences implied by the previous timing diagram
Why Regular Expressions?

- **Regular expressions directly map to NDFA’s**
  - NDFA has one state per identifier in the RE

- **A Circuit can always be constructed directly from RE**
  - One memory element per NDFA state is always sufficient

- **So-- A simple Regular Expression => a Simple Circuit**

![Regular Expression](a,(b* || ~a))

![NDFA state diagram](a, λ, b)

![Circuit](a, ~a)

- **A valid specification may be ‘open’ -- allowing great freedom to optimize the closure or DFA**
Extended Regular Expressions

- We can extend RE’s by adding several operators which keep the language finite, but greatly improve the conciseness:
  - Sequential AND: A&&B both A and B must accept at the same time
  - Sequential NOT: !A -- accepts whenever A is not accepting
  - Exception: A !E B -- B is enabled if A ‘fails’
    - a failure occurs if the input sequence matches no part of Regular Expression A
    - exceptions provide a succinct means of describing how to close a machine
    - although the complement of a DFA is not a state machine, the complement of a RE defined as all sequences which do not match the given set -- is a DFA!
    - consider a protocol decoder machine on a network: if a series of noise impulses arrives at the input, the machine may not recognize specified protocol. Exceptions provide a controlled means of transferring control in such cases.
    - Exceptions allow specification by all sequences not matching the given subset
  - Actions: A, B[start foo] -- provide a symbol to trigger on acceptence
    - Actions provide a means to describe the required output on recognition
“Production Based Specification” (PBS)

- **An extended regular expression language**
  - Additional operators do not violate “1 memory element per symbol” construction
  - Further enhance the conciseness of machine specifications

- **Hierarchical description**
  - Allows reuse of named ‘productions’
  - Provides natural partitions which can be used for circuit optimization
  - productions from concise DAG of sequential operators

- **Boolean function terminals**
  - Use of BDD’s allows arbitrary Boolean functions to be atomic recognition points
  - Boolean terminals form the leaves of the production DAG’s
  - Sequential nodes from the branches

- **Actions allowed on any production acceptance**
  - actions can symbolically describe output behavior or any controlled activity
  - actions are implicitly ordered from most refined to least refined
**PBS Implementation**

- **Currently, PBS supports 2 design styles:**
  1. Controller Generator w. symbolic outputs (BLIF, ATT or other logic form)
     - Output is the controller only
     - Machine outputs correspond to action activation signals
  2. VHDL Code Generator to produce Synopsys® synthesizable VHDL
     - General system specification since actions are arbitrary (1-cycle) behaviors
     - Controller is PBS synthesized, VHDL combinatorial structure
     - Compiler supports VHDL code drop-ins for declarations, initialization
     - Compiler-Compiler format similar to YACC

- **Internal support for action conflict analysis**
  - Since actions are tied to separate event sequences, determining what actions are simultaneously possible is important

- **Intent is to provide an alternative specification/implemention strategy which is compatible with conventional design flows**
Action Details

- Actions annotate productions to describe which contexts are externally observable

- Multiple actions may fire at the same time- if the same action appears in several places, acceptance of any context will fire the action

- Two actions, belonging to different levels of hierarchy of the same tree carry an implicit ordering-- lower levels execute first

```plaintext
#inputs a b
#productions top p1 p2 p3 p4
::
top -> p1, p2*, p3; [a1]
p1 -> a+,~a; [a2]
p2 -> b [a3];
p3 -> (~b)[a2], p4;
p4 -> p1 && p2;
::
```

- Action a2 fires when either p1 or (~b) accepts
- Actions a1, a2 and a3 fire simultaneously when p3 accepts-- a2, a3 logically execute before a1, even though both are in same clock cycle
- although a2 and a3 can fire in the same cycle, they are logically unordered
“Production Based Specification” (PBS)

PBS Code Fragment

::
#input z0 z1 z2
#production p1 p2 p3 p4
::
p1 -> p2 || p3; {action0}
p2 -> .*, (z0 & z1) {action1}, p4;
p3 -> .*, (z0 & z2), (z0 & z2) {action2}, p4;
p4 -> z0+,~z0;
::

Corresponding NDFA

- Code fragment represents a Regular Expression Tree that encodes all event sequences of the NDFA
The Production DAG

::

#input z0 z1 z2
#production p1 p2 p3 p4
::

p1 -> p2 || p3; {action0}
p2 -> .*, (z0 & z1) {action1}, p4;
p3 -> .*, (z0 & z2), (z0 & z2) {action2}, p4;
p4 -> z0+,-z0;
::
Simplification of the Regular Expression

• The PBS is parsed to form a Production DAG and then to a RE tree

• Several minimization rules reduce the number of terminals:
  
  Eg:
  
  \[(A,B) \lor (A,C) => A, (B \lor C)\]
  \[(A,B) \lor (C,B) => (A \lor C), B\]
  \[A \lor A => A\]
  \[(A*)* => A*\]
  \[A,A* => A*,A => A+\]

• Build "Unique Table" for production node sub-trees
  
  - simplifies efficient recognition of equivalent sub-trees
  - allows rule set application by ordered pattern matching hashed node list
  - terminal reduction is directly reflected in latch elimination
Reduced Production Tree

- Reduce the number of terminal nodes
Machine Construction

- Construction is *linear time* traversal of simplified RE tree

- Each RE node has a substitution template which is passed the current context and which returns the resulting context

- The Start or Reset context is passed to the initial node of the traversal

![Diagram of Regular Expression and Circuit](image-url)
Build Function

- Build is the recursive traversal construction algorithm
- Its time complexity is pseudo-linear, including the logic manipulations

```c
Build (node: *n, Boolean function: f(X)) {
    if (n is a terminal function t_j(Z)) {
        create new control point x_i;
        set h(X) = f_i(X, Z) = y_i = f(X) \land t_j(Z);
    } else if (n is concatenation node) {
        g(X) = Build(node->left, f(X));
        h(X) = Build(node->right, g(X));
    } else if (n is sequential and node) {
        g(X) = Build(n->left, f(X));
        h(X) = Build(n->right, f(X));
        h(X) = g(X) \land h(X);
    } else if (...) {
        ... other cases ...
    }
    if (action a_k attached to node) {
        set c_k(X) = h(X);
    }
    return h(X);
}
```


**Node Templates**

### Terminal Function

- $f(X)$
- $t_j(Z)$
- $f(X) \& t_j(Z) \rightarrow x_i = h(X)$

### Concatination

- $f(X)$
- $g(X)$
- $h(X)$
- $f(X) \rightarrow g(X)$
- $g(X) \rightarrow h(X)$
- $f(X) \& g(X) \rightarrow h(X)$
- $f(X) | g(X)$
- $g(X) | h(X)$

### And / Or

- $f(X)$
- $g(X)$
- $h(X)$
- $f(X) \& g(X) \rightarrow h(X)$
- $f(X) | g(X) \rightarrow h(X)$

### Kleene Closure

- $f(X)$
- $g(X, x_t)$
- $h(X)$
- $f(X) \rightarrow g(X, x_t)$
- $f(X) | g(X, x_t) \rightarrow h(X)$
- $x_t \leq f(X) | g(X, x_t)$

---

**Symbols:**
- $f(X)$
- $g(X)$
- $h(X)$
- $t_j(Z)$
- $M_{\text{left}}$
- $M_{\text{right}}$
- $x_i = h(X)$
- $f(X) \& t_j(Z) \rightarrow x_i$
- $f(X) \rightarrow g(X)$
- $g(X) \rightarrow h(X)$
- $f(X) \& g(X) \rightarrow h(X)$
- $f(X) | g(X) \rightarrow h(X)$
- $x_t \leq f(X) | g(X, x_t)$
- $x_t \rightarrow f(X) | g(X, x_t)$
Exception Construction

- Exception machines are constructed simultaneously with the traversal construction of Build. Each template also contains an exception sub-machine and a “working” context construction.

As the traversal progresses, both the valid and complement machines are built, if an exception is needed, the complement machine context is used.
Merging Equivalent States

- **During Construction, identify and merge some equivalent NDFA states**
  - since only a subset of control points have actions, it is possible that many states are output indistinguishable
  - could maintain a list of current DFA states-- re-introducing state-space explosion

- **However, we can often eliminate redundant states without global analysis!**
  - in the figure below, at context $h(X)$, we can not tell if the context was activated by $t_0$ acceptance, $t_2$ acceptance or both-- since there are no actions on these contexts, we don’t care which. A redundant context $h'(X)$ is created and other logic may become redundant and be removed.
Merging Equivalent States

• Create a redundant state bit:
  - For any current context function, \( h(x_1,x_2,...) \), we create a new state bit, \( x_i \) as follows:
    \[ \delta_i(x_1,x_2,...) = h(\delta_1,\delta_2,...) \]
    where each \( \delta_j \) corresponds to state bit \( x_i \), by retiming each \( x_i \).
  - This state bit is redundant, but may allow simplification of the acceptance function:
    \[ h'(X) = x_i \]
  - More importantly, all such contexts are added to the context hash table, where any later bit construction can make use of them

• Context to be removed must not carry an action (observability point)

• Logic commonly eliminated at:
  - Closure and ‘+’ nodes
  - Sequential function nodes (and, or, not)
  - Replicated sub-machines if context allows
**Constructing Machine**

**Table:**

<table>
<thead>
<tr>
<th></th>
<th>Literals</th>
<th>Depth</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PBS</strong></td>
<td>13</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>NOVA</td>
<td>13(16)</td>
<td>3(4)</td>
<td>3</td>
</tr>
<tr>
<td>JEDI</td>
<td>22</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>one-hot</td>
<td>38</td>
<td>9</td>
<td>6</td>
</tr>
</tbody>
</table>
## Encoding Results

<table>
<thead>
<tr>
<th>design example</th>
<th>Current STG states</th>
<th>Clairvoyant min STG states</th>
<th>NOVA</th>
<th>JEDI</th>
<th>one hot</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L</td>
<td>D</td>
<td>R</td>
<td>L</td>
<td>D</td>
</tr>
<tr>
<td>ex1 (Fig. 1)</td>
<td>13</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>22</td>
</tr>
<tr>
<td>mouse</td>
<td>10</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>18</td>
</tr>
<tr>
<td>xymouse</td>
<td>20</td>
<td>3</td>
<td>4</td>
<td>10</td>
<td>36</td>
</tr>
<tr>
<td>mouse2</td>
<td>24</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>36</td>
</tr>
<tr>
<td>xymouse2</td>
<td>48</td>
<td>4</td>
<td>12</td>
<td>50</td>
<td>72</td>
</tr>
<tr>
<td>count0</td>
<td>8</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>16</td>
</tr>
<tr>
<td>qr42</td>
<td>42</td>
<td>6</td>
<td>13</td>
<td>33</td>
<td>77</td>
</tr>
<tr>
<td>i8251ar</td>
<td>67</td>
<td>8</td>
<td>13</td>
<td>15</td>
<td>84</td>
</tr>
<tr>
<td>i2c</td>
<td>115</td>
<td>8</td>
<td>24</td>
<td>70</td>
<td>156</td>
</tr>
<tr>
<td>midi</td>
<td>223</td>
<td>16*</td>
<td>107</td>
<td>107</td>
<td>604</td>
</tr>
<tr>
<td>match1</td>
<td>10</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>match2</td>
<td>21</td>
<td>3</td>
<td>7</td>
<td>15</td>
<td>26</td>
</tr>
<tr>
<td>match4</td>
<td>43</td>
<td>4</td>
<td>16</td>
<td>111</td>
<td>56</td>
</tr>
<tr>
<td>match8</td>
<td>87</td>
<td>4</td>
<td>36</td>
<td>6151</td>
<td>116</td>
</tr>
<tr>
<td>match16</td>
<td>175</td>
<td>5</td>
<td>80</td>
<td>~10^7</td>
<td>236</td>
</tr>
<tr>
<td>match32</td>
<td>351</td>
<td>5</td>
<td>176</td>
<td>~10^{14}</td>
<td>476</td>
</tr>
</tbody>
</table>
Example: Interlocked Pipeline

- Idea: a pipeline in which each stage checks the next stage to see if its context is not stalled

- Want description to only refer to next stage, if possible

- Want ability to interlock (stall some stage) on arbitrary event

- Requires new operator: *marker*

  - A *marker* is a special action which provides a Boolean output that can be tested to determine the current or previous state of other productions
    - allows a production to test the acceptance of any set of simultaneously active productions
    - provides concise non-local access to context of other productions
    - can make description far simpler and more intuitive, but can be overused since:

- Markers allow arbitrary state transition description as well as regular expression format specification and mixed modes.
Interlocked Pipeline Example

- Code format is local -- only the dependence on the next stage is identified

```vhd
moore {} -- mealy machine format => actions available in same cycle as input
::
#input n -- declare only input
#production machine trans it1 it2 it3 it4 it5
#marker i1 i2 i3 i4 i5 i6 -- create observation points for context
#action s0 [s0 <= '1';] -- VHDL actions
#action s1 [s1 <= '1';]
#action s2 [s2 <= '1';]
#action s3 [s3 <= '1';]
#action s4 [s4 <= '1';]
::
machine -> (.* , trans); -- Start transaction each clock
trans -> (it5<i6> + !E)<s0>, (it4<i5> + !E)<s1>, (it3<i4> + !E)<s2>, (it2<i3> + !E)<s3>,
(it1<i2> + !E)<s4><i1>;

it1 -> i1(1); -- 1-delay, this causes <s4> acceptance only on alternate cycles
it2 -> i2(0);
it3 -> i3(0);
it4 -> i4(0);
it5 -> i5(0)|¬n; -- start new transaction only when n is samples true
::
```
Detailed Interlock

- Each stage tries to detect a stall in the next stage, next cycle

- Since each context is activated on arrival of a control point, the construction detects if the next stage is halted (i.e. contains an active control point) if so -- it stalls the current stage.

- Note that every Zero delay marker is part of a combinatorial fan-in network

- Can easily create inconsistent logic -- PBS conservatively warns the user
Interlocked Pipeline-- PBS Encoding/SIS Min Area

- Depth = 5*, States = 48 (24 unique)
- Literals = 24
Interlock Pipeline Comparison

- PBS Machine has 48 states -- Minimal machine has 24 states
- PBS version has 23 literals, depth of 5 (mapped in two-input gates)
- NOVA (PLA optimal) machine has 5 latches, 110 literals, depth of 14
- JEDI machine also has 5 latches, 60 literals, depth of 7
- Retiming Jedi machine increases latches to 12, saves 15% cycle time cost of 90 literals. (Depth is still 7).
- Retimed PBS machine still has 9 latches, but 4 levels -- nearly 2x as fast as best JEDI design, even given unrestricted literals
- PBS design mapped into Willamette technology (Yuji’s mapper): required 3 levels of logic, 45 literals (10 gates, 11 inv) mapped.
- Retimed PBS machine also mapped into 3-levels, 51 literals (12 gates, 10 inv).