Derivation of Monotonic Covers for Standard C Implementation Using STG Unfoldings

Victor Khomenko
Asynchronous Circuits

Asynchronous circuits – no clocks:

😊 Low power consumption
😊 Average-case rather than worst-case performance
😊 Low electro-magnetic emission
😊 Modularity – no problems with the clock skew

😢 Hard to synthesize
CG, gC and stdC architectures
Example: Deriving Equations

\[ \text{Nxt}_z(s) = \text{Code}_z(s) \oplus \text{Out}_z(s) \]

\[ \text{Set}_z / \text{Reset}_z(s) = \begin{cases} 
1 & \text{if } \text{Out}_{z+/z-}(s) = 1 \\
0 & \text{if } \text{Nxt}_z(s) = 0/1 \\
- & \text{otherwise}
\end{cases} \]
Monotonic cover condition

The cover must be entered only via the states enabling the output

Violation

\[ a \lor \overline{b} \lor d \]

\begin{align*}
0100 & \xrightarrow{c^+} 0110 \\
0110 & \xrightarrow{a^-} 1110 \\
1110 & \xrightarrow{d^-} 1111 \\
1111 & \xrightarrow{c^+} 1101 \\
1101 & \xrightarrow{d^+} 1100 \\
1100 & \xrightarrow{b^+} 1000 \\
1000 & \xrightarrow{a^+} 0000 \\
0000 & \xrightarrow{b^+} 0100
\end{align*}
Violation of MC condition

Correct implementation
Strict Set/Reset functions

- Guarantee a correct stdC implementation
- Bad for synthesis (few don’t cares)
- Can be used for overapproximating the support

\[ sSet_z / sReset_z (s) = Out_{z+/z-} (s) \]
Entrance constraints

MC condition: The cover must be entered only via the states enabling the output.

Set_{c}(0110) \Rightarrow Set_{c}(1110)
Set_{c}(1110) \Rightarrow Set_{c}(1111)

Nxt_z(s) = 0
### Example: Deriving Equations

<table>
<thead>
<tr>
<th>Code</th>
<th>( \text{Set}_c )</th>
<th>( \text{Reset}_c )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{abcd} )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>0110</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>0010</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1100</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1110</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1111</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

#### Cover
- \( \overline{abvd} \)
- \( \overline{b} \)

#### Entrance Constraints
- \( \text{Set}_c(0110) \Rightarrow \text{Set}_c(1110) \)
- \( \text{Set}_c(1110) \Rightarrow \text{Set}_c(1111) \)
- \( \emptyset \)

#### Monotonic cover
- \( \overline{abcvd} \)
- \( \overline{b} \)
State Graphs vs. Unfoldings

State Graphs:
- 🤝 Relatively easy theory
- 🤝 Many algorithms
- 🙁 Not visual
- 🙁 State space explosion

Unfoldings:
- 🤝 Alleviate state space explosion
- 🤝 Visual
- 🤝 Proven efficient for model checking
- 🙁 Complicated theory
- 🙁 Relatively few algorithms
Synthesis using unfoldings

Outline of the algorithm for stdC synthesis:

for each output signal $z$

compute minimal supports of $sSet_z/sReset_z$

for each ‘promising’ support $X$

compute the projection of the set of reachable encodings onto $X$ sorting them according to the corresponding value of non-strict $Set_z/Reset_z$

derive the entrance constraints

apply constrained Boolean minimization to the obtained ON- and OFF-sets

choose the best implementation of $z$
CSC properties

The $s\text{CSC}_{X}^{z^+}/s\text{CSC}_{X}^{z^-}$ property: $s\text{Set}_{z}/s\text{Reset}_{z}$ is a well-defined Boolean function of projection of the encoding of the current state on set of signals $X$; i.e., $X$ is a support.
CSC conflicts

States $M'$ and $M''$ are in $s\text{CSC}_{X}^{Z^+} / s\text{CSC}_{X}^{Z^-}$ conflict if

- $\text{Codex}_x(M') = \text{Codex}_x(M'')$ for all $x \in X$ and
- $F(M') \neq F(M'')$

where $F = \text{sSet}_Z / \text{sReset}_Z$

$F$ can be expressed as a Boolean function with support $X$ iff there are no conflicts of the corresponding type
Example

\[
\begin{array}{cccc}
0100 & \rightarrow & b+ & \rightarrow & a+ & \rightarrow & 1000 \\
\uparrow & & c+ & \uparrow & \downarrow & \downarrow & \downarrow & \downarrow \\
0110 & \rightarrow & b- & \rightarrow & 0010 & \rightarrow & 1100 & \rightarrow & 1111 & \rightarrow & 1101 \\
\downarrow & & a- & \downarrow & \rightarrow & d- & \rightarrow & c+ & \rightarrow & \end{array}
\]

\[s\text{CSC}^c_{\{a,b,d\}} \text{ conflict, but ok for gC}\]
Example: $\text{sCSC}_{X}^{z^+}$ conflict in prefix

- $C'$
- $C''$

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$\text{Code}(C')$</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- $X=\{a,b,d\}$, $\text{Code}_{X}(C')=\text{Code}_{X}(C'')$

$s\text{CSC}^{c^+}$ conflict!

$\text{Out}_{c^+}(C')=1 \neq \text{Out}_{c^+}(C'')=0$

$\text{Nxt}_{c}(C')=1 = \text{Nxt}_{c}(C'')$ ok for gC
Computing non-supports

- Using unfoldings, it is possible to construct a Boolean formula $\text{CSC}^F(X,\ldots)$ such that $\text{CSC}^F(X,\ldots)[Y/X]$ is satisfiable iff $Y$ is not a support of $F=s\text{Set}_Z/s\text{Reset}_Z$.

- The projection of the set of satisfying assignments of $\text{CSC}^Z(X,\ldots)$ onto $X$ is the set of all non-supports of $F$ (it is sufficient to compute the maximal elements of this projection).

Need to know how to compute projections!
Example: projections

\[ \varphi = (a \lor b)(\neg a \lor \neg b)(c \lor d \lor e) \]

\[ a \oplus b \]

\[
\begin{array}{cccccc}
0 & 1 & 0 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 1 & 1 & 1 \\
0 & 1 & 1 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 1 \\
0 & 1 & 1 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 & 1 & 1 \\
1 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 \\
1 & 0 & 1 & 1 & 0 & 1 \\
1 & 0 & 1 & 1 & 1 & 1 \\
1 & 0 & 1 & 1 & 1 & 1 \\
1 & 0 & 1 & 1 & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{ccc}
a & b & c \\
0 & 1 & 0 \\
0 & 1 & 1 \\
1 & 0 & 0 \\
1 & 0 & 1 \\
\end{array}
\]

\[
\begin{array}{ccc}
a & b & c \\
0 & 1 & 1 \\
1 & 0 & 1 \\
\end{array}
\]

\[
\begin{array}{ccc}
a & b & c \\
0 & 1 & 0 \\
1 & 0 & 0 \\
1 & 0 & 1 \\
\end{array}
\]

\[
\begin{array}{ccc}
a & b & c \\
0 & 1 & 0 \\
1 & 0 & 0 \\
\end{array}
\]
Computing projections

\[ \phi = (a \lor b)(\overline{a} \lor \overline{b})(c \lor d \lor e)(a \lor b \lor c)(\overline{a} \lor \overline{b} \lor \overline{c})(a \lor b \lor c)(\overline{a} \lor b \lor \overline{c}) \]

\[ \text{Proj}_{\{a, b, c\}} \phi \]

\[
\begin{array}{cccccc}
  a & b & c & d & e \\
  0 & 1 & 0 & 0 & 1 \\
  0 & 1 & 1 & 0 & 0 \\
  1 & 0 & 0 & 0 & 1 \\
  1 & 0 & 1 & 0 & 0 \\
  UNSAT
\end{array}
\]

- Incremental SAT
Computing supports

• The set of maximal non-supports is known

• The set of supports:
  \[ \{ Y \mid Y \not\subseteq X, \text{ for all maximal non-supports } X \} \]

• The problem can be reduced to computing the compliment of a unate Boolean function

• For example, let \{\{a,b,c\},\{a,b,d\},\{a,c,d\},\{b,c,d\}\} be the set of maximal non-supports

• The corresponding characteristic function is
  \[ \neg a + \neg b + \neg c + \neg d \]

• Its complement is \(abcd\), so the set of minimal supports is \{\{a,b,c,d\}\}
Computing entrance constraints

• Given a support $X$, construct a Boolean formula $EC^F(X,\ldots)$ such that $EC^F(X,\ldots)[Y/X]$ is satisfiable iff there is a reachable state from which it is possible to illegally enter the cover with the encoding projection $Y$

• Use Incremental SAT to compute the set of entrance constraints (on each step, a clause ruling out all the satisfying assignments which would result in the computed entrance constraint is added)
Computing the set/reset covers

• Compute the projection of the set of reachable encodings onto the given support X partitioning them according to the corresponding value of $\text{Set}_z/\text{Reset}_z$
• Apply conditional (binate) Boolean minimization to this projection and the entrance constraints
Optimizations

😊 Triggers belong to every support – significantly improves the efficiency

😊 Further optimizations are possible for certain net subclasses, e.g. unique-choice nets
Experimental Results

• Unfoldings of STGs are almost always small in practice and thus well-suited for synthesis
• Huge memory savings
• Dramatic speedups
• Every valid speed-independent solution can be obtained using this method, so no loss of quality
• We can trade off quality for speed (e.g. consider only minimal supports): in our experiments, the solutions are the same as Petrify’s (up to Boolean minimization)
• Multiple implementations produced
Thank you!
Any questions?