The Royal Academy of Engineering

Newcastle University

Derivation of Monotonic Covers for Standard C Implementation Using STG Unfoldings

Victor Khomenko

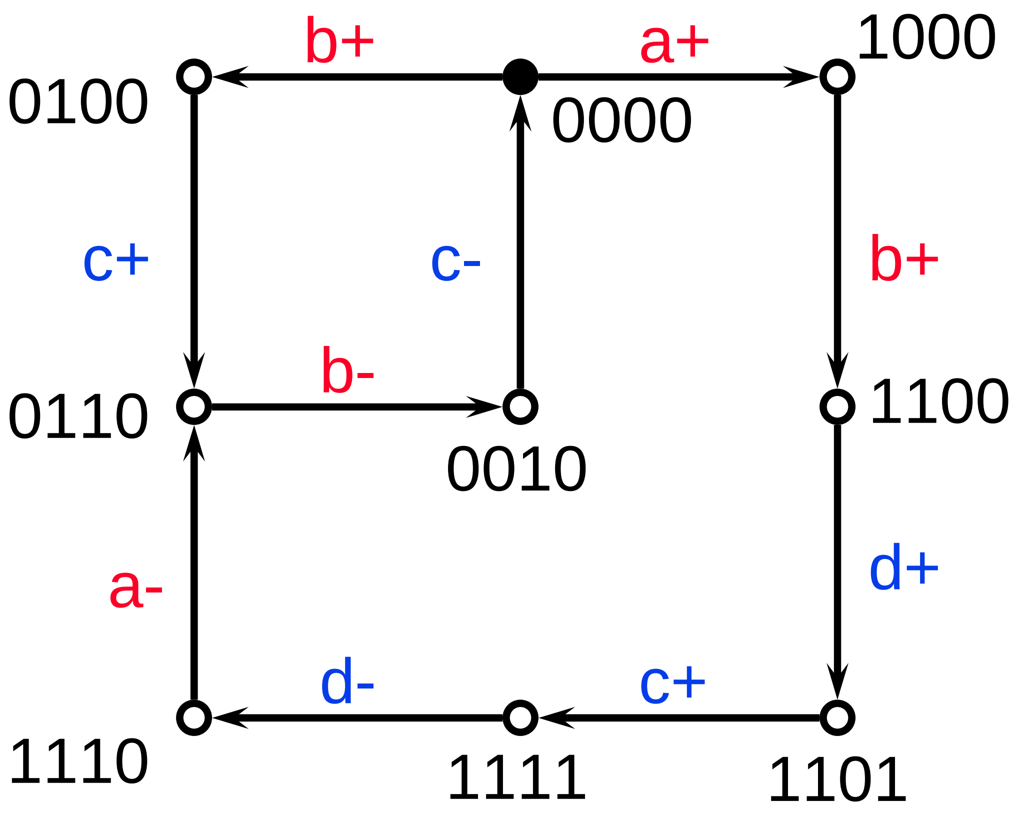
Asynchronous Circuits

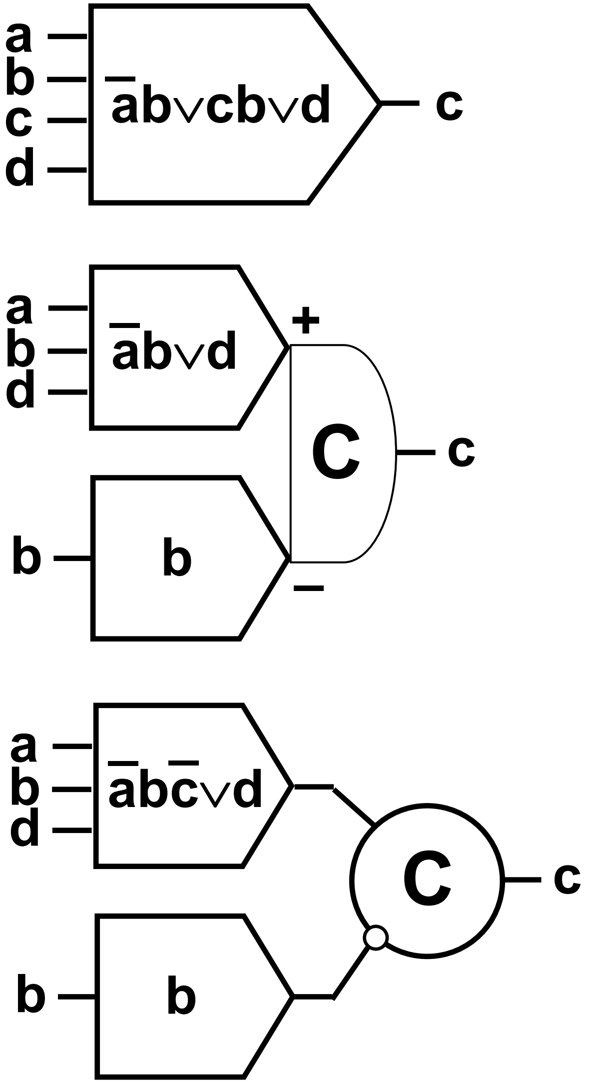
Asynchronous circuits – no clocks:

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

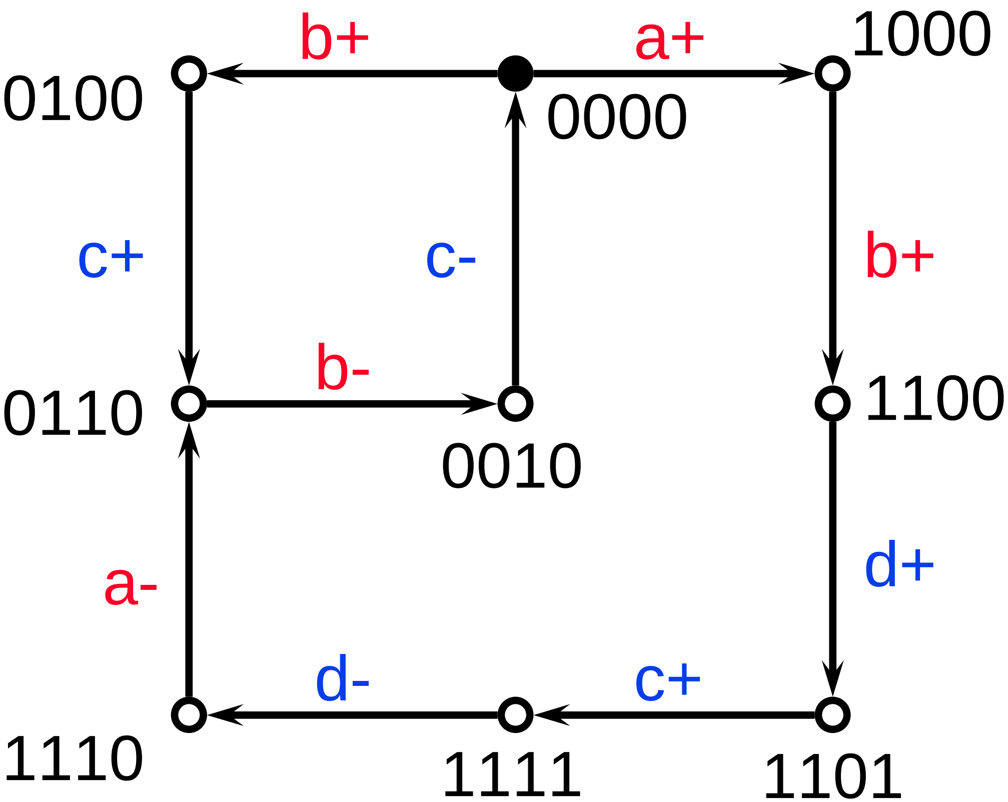
- ☹ Hard to synthesize

CG, gC and stdC architectures





Example: Deriving Equations



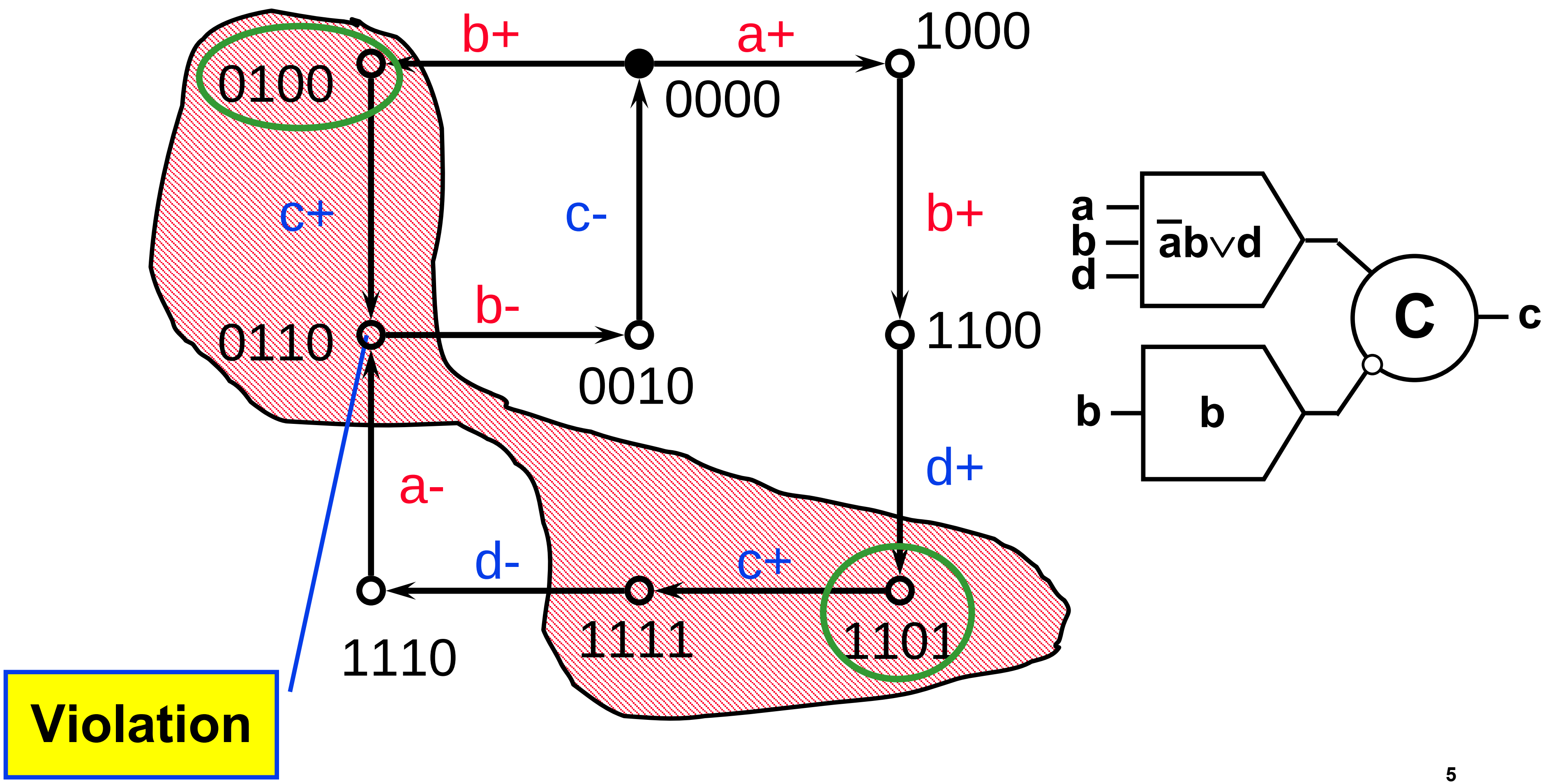
Code	Nxt_c	Set_c	$Reset_c$
0100	1	1	0
0000	0	0	-
1000	0	0	-
0110	1	-	0
0010	0	0	1
1100	0	0	-
1110	1	-	0
1111	1	-	0
1101	1	1	0
Eqn	$\bar{a}b \vee cb \vee d$	$\bar{a}b \vee d$	$\bar{b}$

$$Nxt_z(s) = Code_z(s) \oplus Out_z(s)$$

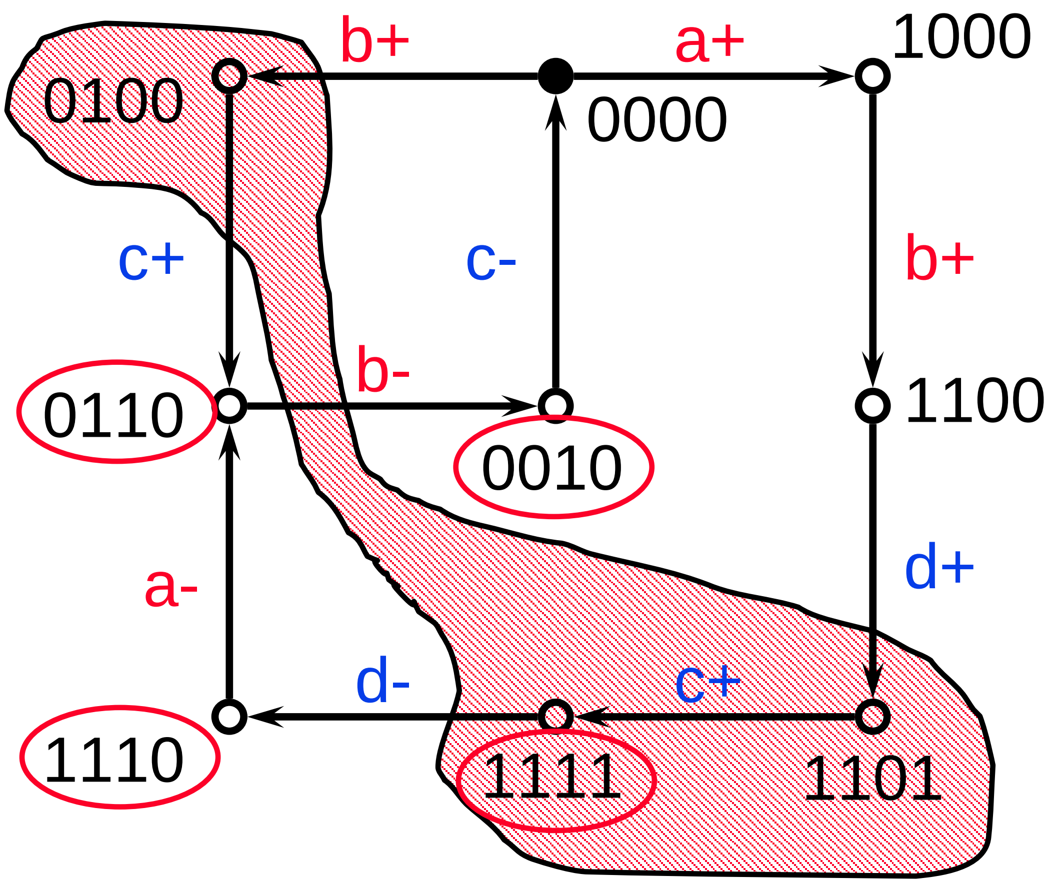
$$Set_z / Reset_z(s) = \begin{cases} 1 & \text{if } Out_{z+/z-}(s) = 1 \\ 0 & \text{if } Nxt_z(s) = 0/1 \\ - & \text{otherwise} \end{cases}$$

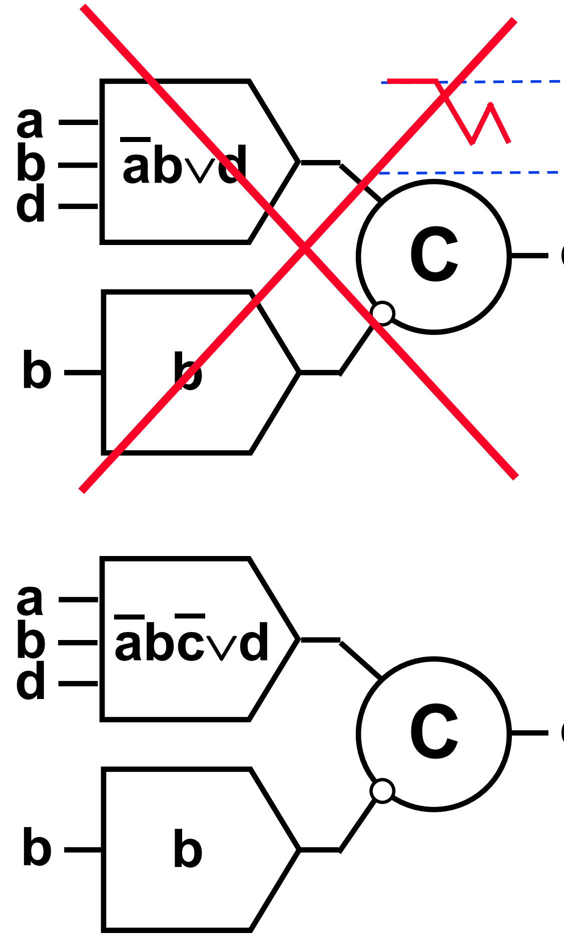
Monotonic cover condition

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



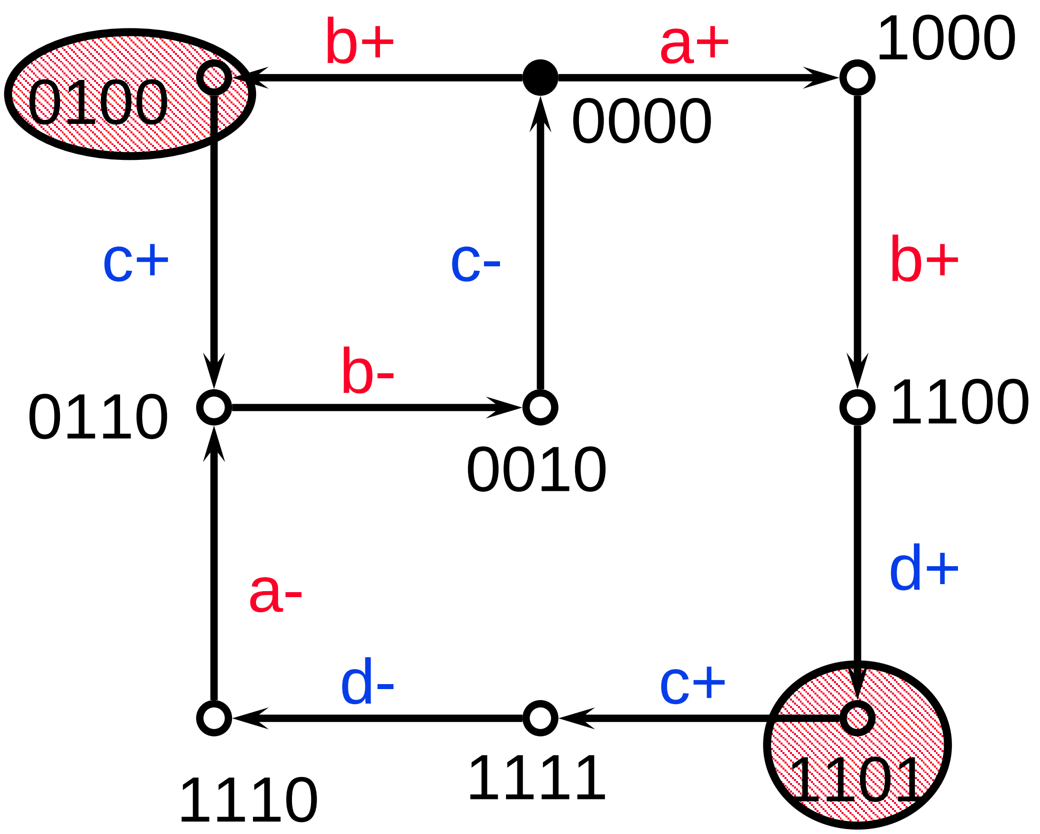
Violation of MC condition





Correct implementation

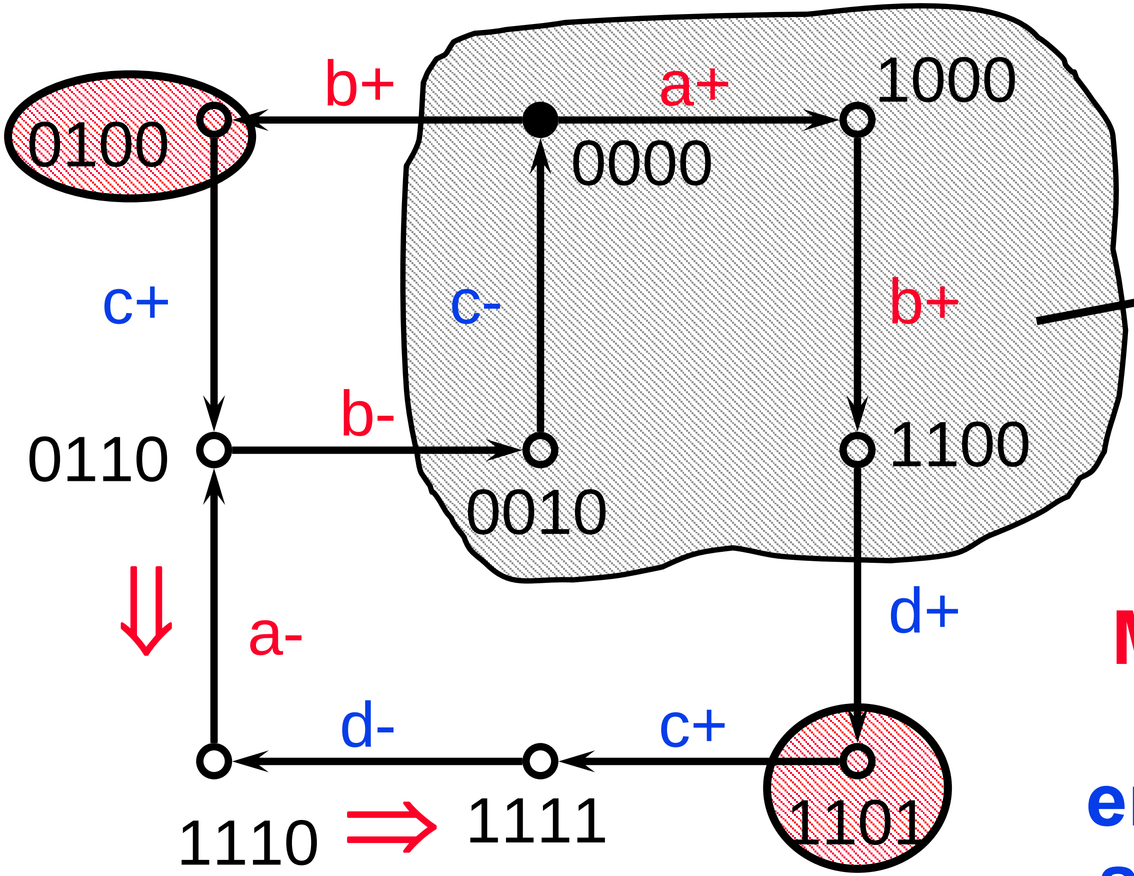
Strict Set/Reset functions



- 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



$$Nxt_z(s) = 0$$

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

Set$_c$(0110)⇒Set$_c$(1110)

Set$_c$(1110)⇒Set$_c$(1111)

Example: Deriving Equations

Code abcd	Set_c	$Reset_c$
0100 0000 1000 0110 0010 1100 1110 1111 1101	1 0 0 - 0 0 - - 1	0 - - 0 1 - 0 0 0
Cover	$\overline{a}b \vee d$	$\overline{b}$
Entrance Constraints	$Set_c(0110) \Rightarrow Set_c(1110)$ $Set_c(1110) \Rightarrow Set_c(1111)$	$\varnothing$
Monotonic cover	$\overline{a}b\overline{c} \vee d$	$\overline{b}$

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 $sCSC_X^{z+}$ /$sCSC_X^{z-}$ property: $sSet_z$/$sReset_z$ 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

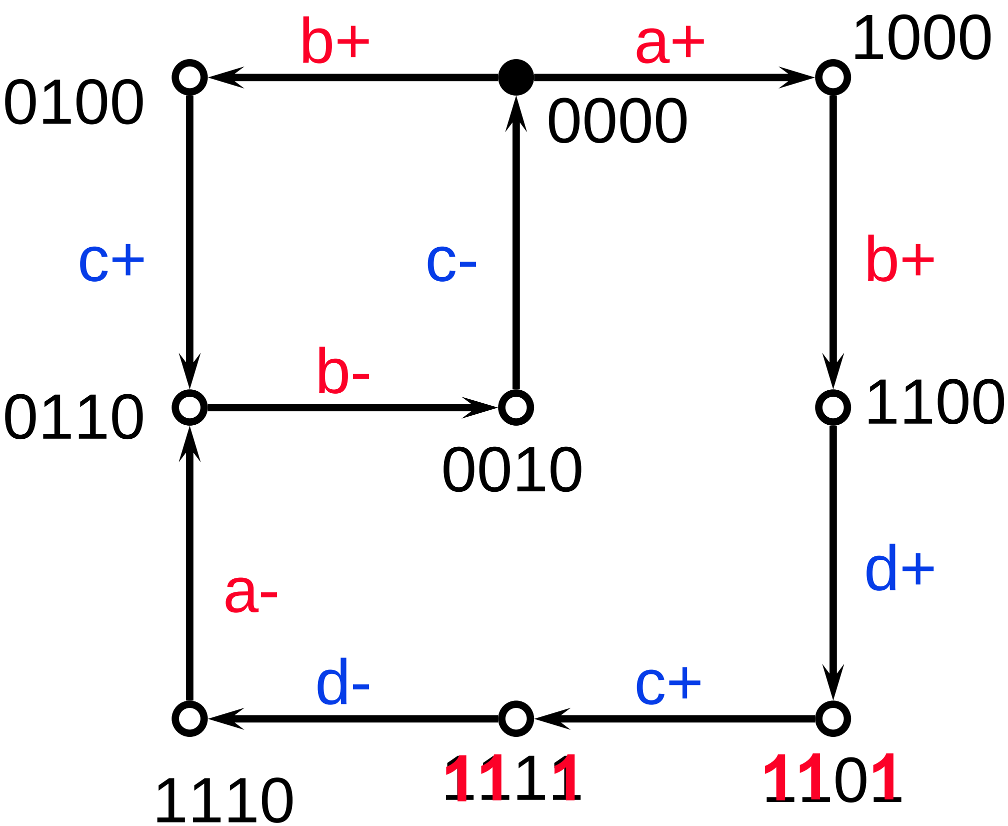
States M' and M'' are in $sCSC_X^{z+}$ / $sCSC_X^{z-}$ conflict if

- $Code_x(M')=Code_x(M'')$ for all $x \in X$ and
- $F(M') \neq F(M'')$

where $F=sSet_z/sReset_z$

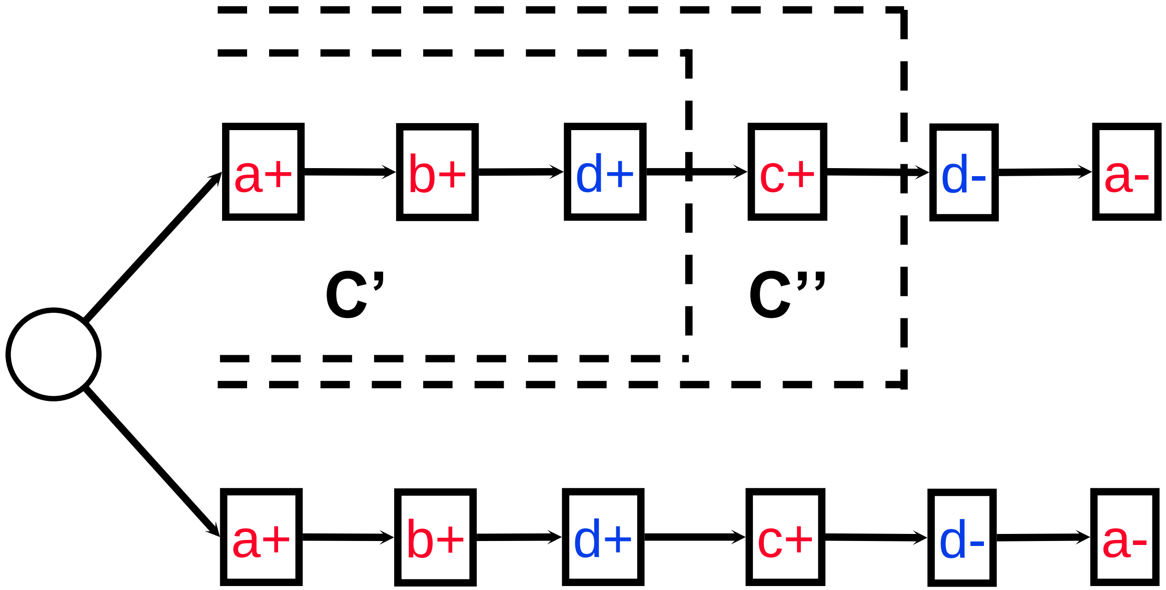
F can be expressed as a Boolean function with support X iff there are no conflicts of the corresponding type

Example



$\mathbf{sCSC^{c+}_{\{a,b,d\}}}$ **conflict, but ok for gC**

Example: $sCSC^{z+}_{X}$ conflict in prefix



	a	b	c	d
Code(C')	1	1	0	1
Code(C'')	1	1	1	1

$Out_{c+}(C')=1 \neq Out_{c+}(C'')=0$

sCSC^{c+} conflict!

$Nxt_{c}(C')=1=Nxt_{c}(C'')$

ok for gC

$X=\{a,b,d\}$, $Code_{X}(C')=Code_{X}(C'')$

Computing non-supports

- Using unfoldings, it is possible to construct a Boolean formula $CSC^F(X,\ldots)$ such that $CSC^F(X,\ldots)[Y/X]$ is satisfiable iff **Y** is *not* a support of $F{=}sSet_z/sReset_z$
- The projection of the set of satisfying assignments of $CSC^z(X,\ldots)$ 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

$$\varphi = \overbrace{(a \vee b)(\neg a \vee \neg b)}^{a \oplus b}(c \vee d \vee e)$$

a	b	c	d	e
0	1	0	0	1
0	1	0	1	0
0	1	0	1	1
0	1	1	0	0
0	1	1	0	1
0	1	1	1	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	0	1	1
1	0	1	0	0
1	0	1	0	1
1	0	1	1	0
1	0	1	1	1

$\mathrm{Proj}_{\{a,b,c\}}\ \varphi$

a	b	c
0	1	0
0	1	1
1	0	0
1	0	1

max $\mathrm{Proj}_{\{a,b,c\}}\ \varphi$

a	b	c
0	1	1
1	0	1

min $\mathrm{Proj}_{\{a,b,c\}}\ \varphi$

a	b	c
0	1	0
1	0	0

Computing projections

$$\varphi=\overbrace{(a\vee b)(\bar{a}\vee \bar{b})}^{a \oplus b}(c\vee d\vee e)(a\vee \bar{b}\vee c)(a\vee \bar{b}\vee \bar{c})(\bar{a}\vee b\vee c)(\bar{a}\vee b\vee \bar{c})$$

$\mathrm{Proj}_{\{a,b,c\}}\ \varphi$

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

- Incremental SAT

Computing supports

- The set of maximal non-supports is known
- The set of supports:

  $\{ Y \mid Y \not\subseteq X$, 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 $Set_z/Reset_z$
- 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?