Symmetry Reduction with STE Model Checking

Ashish Darbari

Oxford University Computing Lab
Oxford

20 Dec 2005

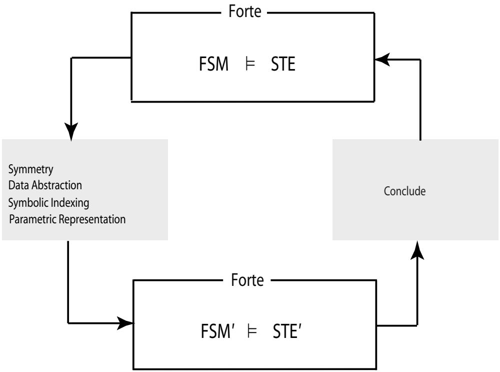
Motivation

- Hardware designs have symmetry.
- For circuit designs that have symmetry, we aim to exploit reduction techniques that can make use of the symmetry property, to reduce the size of STE verification task needed for complete verification of that circuit design.

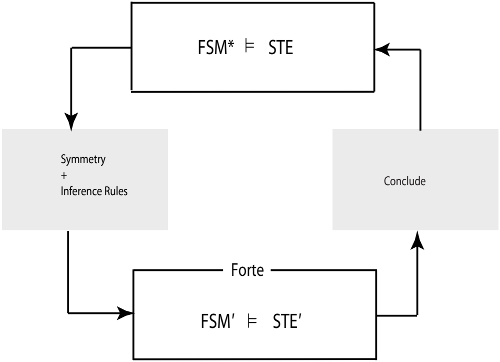
Motivation

- Hardware designs have symmetry.
- For circuit designs that have symmetry, we aim to exploit reduction techniques that can make use of the symmetry property, to reduce the size of STE verification task needed for complete verification of that circuit design.

What we want to achieve?



Proposed solution



In a nutshell

Two key components

- Discover symmetry
- Do property reduction

In a nutshell

Two key components

- Discover symmetry
- Do property reduction

Discover Symmetry

Key issues

- Structural symmetry
- How to find them?
- What have symmetries in circuits got to do with STE?

Discover Symmetry

Key issues

- Structural symmetry
- How to find them?
- What have symmetries in circuits got to do with STE?

Discover Symmetry

Key issues

- Structural symmetry
- How to find them?
- What have symmetries in circuits got to do with STE?

Structural symmetry

- We are interested in the symmetry amongst groups of wires. Wires are kept together in a group. Every wire in the group is treated in exactly the same way.
- If such is the case then the input-output behaviour of the circuit remains independent under permutations of its input and output groups of wires. This kind of symmetry is what we refer to as structural symmetry.

How to find them?

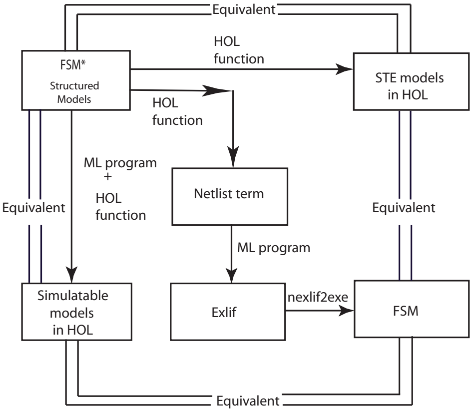
- We want to capture symmetry in the structure of a circuit in its description right at the level of design, which means *structured high-level design via a structured data type.*
- Symmetry discovery then reduces to type checking. This idea by itself is not new — it has been around for a while in the model checking community. But for STE this is the very first time.
- We propose a structured data type of models, a type system for designing symmetric circuits and prove a type soundness theorem that says that if a circuit is well-behaved with respect to the typing rules then it has structural symmetry.

Relation of Symmetry with STE

Symmetric models and STE

Symmetry in circuit models is mirrored by symmetry in STE properties. We formalise this by a theorem that articulates this connection.

Going from FSM* to FSM′



Property Reduction – I

Two key issues

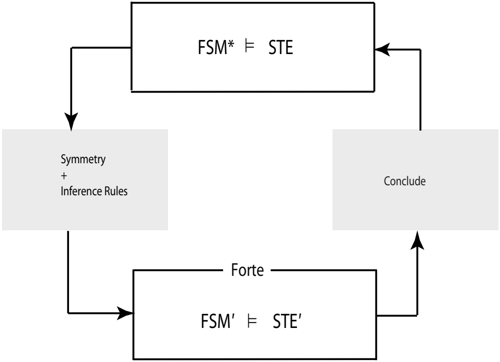
- We need to figure out the path from STE to STE′
- Verifying STE′ and deducing that STE has been done

We present a novel set of inference rules that will help achieve both the above targets. Inference rules can help decompose STE to STE′, if used like tactics, and help compose the overall correctness statement when used in the forward direction.

Property Reduction – II

- Symmetry in circuit models lets us partition the decomposed STE properties into equivalence classes.
- We verify only the representatives and conclude that the other members of the same equivalence classes have been verified as well by way of deduction rather than explicit STE verification.

Proposed solution revisited



FSM*

Designing the FSM*

Important issues regarding the design of FSM*

- design of a type of structured models
- define type checking rules
- keep the design of the type system simple
- prove the type soundness lemma
- figure out the path from FSM* to FSM$'$

Designing the FSM*

Important issues regarding the design of FSM*

- design of a type of structured models
- define type checking rules
- keep the design of the type system simple
- prove the type soundness lemma
- figure out the path from FSM* to FSM$'$

Designing the FSM*

Important issues regarding the design of FSM*

- design of a type of structured models
- define type checking rules
- keep the design of the type system simple
- prove the type soundness lemma
- figure out the path from FSM* to FSM$'$

Designing the FSM^*

Important issues regarding the design of FSM^*

- design of a type of structured models
- define type checking rules
- keep the design of the type system simple
- prove the type soundness lemma
- figure out the path from FSM^* to FSM'

Designing the FSM*

Important issues regarding the design of FSM*

- design of a type of structured models
- define type checking rules
- keep the design of the type system simple
- prove the type soundness lemma
- figure out the path from FSM* to FSM$'$

FSM* – Type of structured models

type of structured models

$$c : \textit{bool list list} \;\rightarrow\; \textit{bool list list} \;\rightarrow\; \textit{bool list list}$$

- want to model a collection of bit (Boolean) values
- treat them in a special way
- model the collection of values at wires by lists of Boolean value
- if there are several such bundles then we employ a list of Boolean lists modelling the inputs and outputs of circuits
- first argument acts as a placeholder for non-symmetric input bundles and the second argument of the circuit type denotes the symmetric input bundles. The third argument denotes the output bundles.

FSM* – Type of structured models

type of structured models

$$c : bool\ list\ list \ \rightarrow\ bool\ list\ list \ \rightarrow\ bool\ list\ list$$

- want to model a collection of bit (Boolean) values
- treat them in a special way
- model the collection of values at wires by lists of Boolean value
- if there are several such bundles then we employ a list of Boolean lists modelling the inputs and outputs of circuits
- first argument acts as a placeholder for non-symmetric input bundles and the second argument of the circuit type denotes the symmetric input bundles. The third argument denotes the output bundles.

FSM* – Type of structured models

type of structured models

$$c : bool\ list\ list \rightarrow bool\ list\ list \rightarrow bool\ list\ list$$

- want to model a collection of bit (Boolean) values
- treat them in a special way
- model the collection of values at wires by lists of Boolean value
- if there are several such bundles then we employ a list of Boolean lists modelling the inputs and outputs of circuits
- first argument acts as a placeholder for non-symmetric input bundles and the second argument of the circuit type denotes the symmetric input bundles. The third argument denotes the output bundles.

FSM* – Type of structured models

type of structured models

$$c : \textit{bool list list} \;\rightarrow\; \textit{bool list list} \;\rightarrow\; \textit{bool list list}$$

- want to model a collection of bit (Boolean) values
- treat them in a special way
- model the collection of values at wires by lists of Boolean value
- if there are several such bundles then we employ a list of Boolean lists modelling the inputs and outputs of circuits
- first argument acts as a placeholder for non-symmetric input bundles and the second argument of the circuit type denotes the symmetric input bundles. The third argument denotes the output bundles.

FSM* – Type of structured models

type of structured models

$$c : bool\ list\ list \ \rightarrow\ bool\ list\ list\ \rightarrow\ bool\ list\ list$$

- want to model a collection of bit (Boolean) values
- treat them in a special way
- model the collection of values at wires by lists of Boolean value
- if there are several such bundles then we employ a list of Boolean lists modelling the inputs and outputs of circuits
- first argument acts as a placeholder for non-symmetric input bundles and the second argument of the circuit type denotes the symmetric input bundles. The third argument denotes the output bundles.

FSM* – Some useful functions I

$$\vdash \; hd \; (h :: t) \; = \; h$$

$$\vdash \; tl \; (h :: t) \; = \; t$$

$$\begin{aligned} &\vdash \; el \; 0 \; l \; = \; hd \; l \\ &\wedge \; el \; (n+1) \; l \; = \; el \; n \; (tl \; l) \end{aligned}$$

$$\begin{aligned} &\vdash \; append \; [\,] \; l \; = \; l \\ &\wedge \; append \; (x :: l_1) \; l_2 \; = \; (x :: (append \; l_1 \; l_2)) \end{aligned}$$

FSM* – Some useful functions II

$$\begin{array}{ll} \vdash & map2\ f\ [\,]\ [\,]\ =\ [\,] \\ \wedge & map2\ f\ (h_1 :: t_1)\ (h_2 :: t_2)\ =\ f\ h_1\ h_2 :: map2\ f\ t_1\ t_2 \end{array}$$

$$\begin{array}{ll} \vdash & foldr\ f\ e\ [\,]\ =\ e \\ \wedge & foldr\ f\ e\ (x :: l)\ =\ f\ x\ (foldr\ f\ e\ l) \end{array}$$

FSM* – Some useful functions III

$$\begin{array}{ll} \vdash & (drop\ \ 0\ \ l\ \ =\ \ tl\ \ l) \\ \wedge & (drop\ \ (i+1)\ \ l\ \ =\ \ drop\ \ i\ \ (tl\ \ l)) \end{array}$$

$$\begin{array}{ll} \vdash & (take\ \ 0\ \ l\ \ =\ \ tl\ \ l) \\ \wedge & (take\ \ (i+1)\ \ (x :: xs)\ \ =\ \ (x :: (take\ \ i\ \ xs))) \end{array}$$

$$\begin{array}{ll} \vdash & (insert\ \ elem\ \ i\ \ lst\ \ = \\ & \quad append(take\ \ i\ \ lst)(elem :: (drop\ \ i\ \ lst))) \end{array}$$

FSM* – Level 0 functional blocks

$$\vdash \quad id \quad = \quad \lambda inp : bool\ list.\ inp$$

$$\vdash \quad f \quad \circ \quad g \quad = \quad (\lambda x.\ f\ (g\ x))$$

$$\begin{array}{l} \vdash \quad (map\ \ f\ \ [\,] \quad = \quad [\,]) \\ \wedge \quad (map\ \ f\ \ (h :: t) \quad = \quad f\ \ h :: map\ \ f\ \ t) \end{array}$$

$$\begin{array}{l} \vdash \quad fold\ \ f\ \ (c : bool\ list\ \longrightarrow bool\ list) \quad = \\ \qquad \lambda inp.\ \ [foldr\ \ f\ \ (hd\ (c\ inp))\ (tl\ (c\ inp))] \end{array}$$

FSM* – Safe functional blocks

$$\frac{}{\mathit{safe}\ \ \mathit{id}}$$

$$\frac{f : \mathit{bool} \ \longrightarrow\ \mathit{bool}}{\mathit{safe}\ \ (\mathit{map}\ \ f)}$$

$$\frac{\mathit{safe}\ \ c \qquad f : \mathit{bool} \ \longrightarrow\ \mathit{bool} \ \longrightarrow\ \mathit{bool}}{\mathit{safe}\ \ (\mathit{fold}\ f\ c)}$$

$$\frac{\mathit{safe}\ \ c_1 \qquad \mathit{safe}\ \ c_2}{\mathit{safe}\ \ (c_1 \circ\ c_2)}$$

FSM* – The Function Swap

$$
\begin{aligned}
\vdash \quad & swap \;\; (i, j) \;\; lst \quad = \\
& \quad if \;\; (length\, lst \;\; > \;\; i) \;\; \wedge \;\; (length\, lst \;\; > \;\; j)) \\
& \qquad then \;\; (insert \; (el \; j \; lst) \;\; i \;\; (insert \; (el \; i \;\; lst) \;\; j \;\; lst)) \\
& \qquad\quad else \;\; lst
\end{aligned}
$$

FSM* – Symmetric Functional Blocks

$$\vdash \; sym\ c \;=\; \forall inp\ i\ j.\ (c\ \ (swap\ (i,j)\ inp) \;=\; swap\ (i,j)\ (c\ inp))$$

Level 0 safety lemma

$$\vdash\ \ \forall c.\ safe\ \ c\ \supset\ sym\ \ c$$

FSM* – Helper functions

Buses of equal length

$$\vdash \; CheckLength\ inp \;=\; \forall l.\ l \in inp \supset \forall m.\ m \in inp \supset \exists k.\ (length\ l = k) \wedge (length\ m = k)$$

Associativity and Commutativity

$$\vdash \; comm\ f \;=\; \forall xy.\ f\ x\ y = f\ y\ x$$

$$\vdash \; assoc\ f \;=\; \forall xyz.\ f\ x\ (f\ y\ z) = f\ (f\ x\ y)\ z$$

Constructing symmetric circuits – Level I

$$\vdash \quad Null \quad = \quad \lambda inp.\ [\,]$$

$$\vdash \quad Id \quad = \quad \lambda inp : (bool\ list)\ list.\ inp$$

$$\vdash \quad (c1 \mid\mid c2) \quad = \quad \lambda sym.\ if\ \ CheckLength\ \ (append\ (c1\ sym)(c2\ sym)) \\ then\ \ append\ \ (c1\ sym)(c2\ sym)\ \ else\ \ [\,]$$

$$\vdash \quad Fork\ \ c \quad = \quad \lambda sym.\ \ append\ (c\ sym)\ (c\ sym)$$

Constructing symmetric circuits – Level I

$$\vdash \mathit{Select}\ n\ c \;=\; \lambda sym.\ \mathit{if}\ (\mathit{length}(c\ sym)\ >\ n)\\ \mathit{then}\ \ [el\ n\ (c\ sym)]\ \ \mathit{else}\ [\,]$$

$$\vdash \mathit{Tail}\ c \;=\; \lambda sym.\ \mathit{if}\ (\mathit{length}(c\ sym))\ >\ 1\\ \mathit{then}\ \ tl\ (c\ sym)\ \ \mathit{else}\ [\,]$$

$$\vdash \mathit{Bitwise}\ f\ c \;=\; \lambda sym.\ \mathit{if}\ (\mathit{length}(c\ sym)\ >\ 0)\\ \mathit{then}\ \ [\mathit{foldr}\ (\mathit{map2}\ \ f)(hd\ (c\ sym))(tl\ (c\ sym))\,]\\ \mathit{else}\ [\,]$$

Typing rules for symmetric circuits – I

$$\frac{}{SS\ \ Null}$$

$$\frac{}{SS\ \ Id}$$

$$\frac{SS\ \ c}{SS\ \ (map\ \ c)}$$

$$\frac{SS\ \ c_1 \qquad SS\ \ c_2}{SS\ \ (c_1 \circ\ c_2)}$$

Typing rules for symmetric circuits – II

$$\frac{SS\ \ c_1 \qquad SS\ \ c_2}{SS\ \ (c_1\ \|\ \ c_2)}$$

$$\frac{SS\ \ c}{SS\ \ (Fork\ \ c)}$$

$$\frac{SS\ \ c \qquad n : num}{SS\ \ (Select\ \ n\ \ c)}$$

$$\frac{SS\ \ c}{SS\ \ (Tail\ \ c)}$$

$$\frac{SS\ \ c \qquad assoc\ f \qquad comm\ f \qquad f : bool\ \rightarrow\ bool\ \rightarrow\ bool}{SS\ \ (Bitwise\ f\ \ c)}$$

Definition of symmetry

Symmetry

$$\begin{aligned} Sym\ \ c \ \ &\triangleq\ \ \forall inp.\ CheckLength\ \ inp\ \ \supset \\ &\qquad \forall i\, j.\ \ map(swap(i,j))(c\ inp) \\ &\qquad\qquad = \\ &\qquad\qquad c\ (map(swap(i,j))\ inp) \end{aligned}$$

Type Soundness Theorem

Structurally safe implies symmetry

$$\vdash \forall c.\ SS\ \ c \ \supset\ Sym\ \ c$$

Validating circuits

$$\vdash \quad \mathit{Validate}\ (c : \mathit{bool\ list\ list} \rightarrow \mathit{bool\ list\ list} \rightarrow \mathit{bool\ list\ list}) \\ = \quad \forall \mathit{nsym}.\ \mathit{SS}\ (c\ \mathit{nsym})$$

Validated circuits have symmetry

$$\vdash \quad \forall c.\ \mathit{Validate}\ c\ \supset\ \forall \mathit{nsym}.\ \mathit{Sym}\ (c\ \mathit{nsym})$$

Adding time to combinational layer

Abstractions of delay elements

rising edge latch

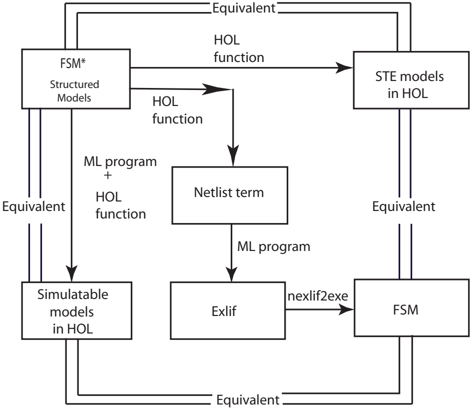
$$DEL\ (clk : bool) \ \triangleq\ \lambda inp : bool.\, inp$$

active high latch

$$AH\ (clk : bool) \ \triangleq\ \lambda inp : bool.\, inp$$

Note that structurally they are equivalent, the behaviours are different and these get interpreted for simulation in HOL, by semantic functions.

FSM* to FSM$'$


Equivalent
HOL
function
FSM*
Structured
Models
STE models
in HOL
HOL
function
ML program
+
HOL
function
Netlist term
Equivalent
Equivalent
ML program
Simulatable
models
in HOL
Exlif
nexlif2exe
FSM
Equivalent

STE Theory

States and sequences

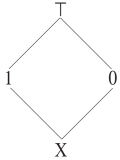
States and sequences

$$s : string \longrightarrow bool \times bool$$
$$\sigma : num \longrightarrow string \longrightarrow bool \times bool$$

Suffix of a sequence

$$\sigma_i \triangleq \lambda t\, n.\, \sigma\,(t + i)\, n$$

Information Ordering



Information ordering on states

$$s_1 \mathrel{\dot{\sqsubseteq}} s_2 \quad \triangleq \quad \forall n : string.\ s_1\ n \sqsubseteq s_2\ n$$

Information ordering on sequences

$$\sigma_1 \mathrel{\ddot{\sqsubseteq}} \sigma_2 \quad \triangleq \quad \forall t : num. \forall n : string.\ \sigma_1\ t\ n \sqsubseteq \sigma_2\ t\ n$$

Circuit models

STE Models – Implemented as FSM in Forte

$$\mathcal{M} : (string \rightarrow bool \times bool) \rightarrow (string \rightarrow bool \times bool)$$

Monotonicity

$$Monotonic\ \mathcal{M} \triangleq \forall s\ s'.\ (s \mathrel{\dot{\sqsubseteq}} s') \supset ((\mathcal{M}\ s) \mathrel{\dot{\sqsubseteq}} (\mathcal{M}\ s'))$$

Syntax of STE formulas

$$
\begin{aligned}
f \triangleq\ & n \text{ is } 0 \\
& \mid n \text{ is } 1 \\
& \mid f \text{ and } g \\
& \mid f \text{ when } P \\
& \mid \mathsf{N}\, f
\end{aligned}
$$

Semantics of STE

$$
\begin{array}{lcl}
(\phi, \sigma) \models n \text{ is } 0 & \triangleq & 0 \sqsubseteq \sigma\, 0\, n \\
(\phi, \sigma) \models n \text{ is } 1 & \triangleq & 1 \sqsubseteq \sigma\, 0\, n \\
(\phi, \sigma) \models f_1 \text{ and } f_2 & \triangleq & (\phi, \sigma) \models f_1 \wedge (\phi, \sigma) \models f_2 \\
(\phi, \sigma) \models f \text{ when } P & \triangleq & (\phi \models P) \supset (\phi, \sigma) \models f \\
(\phi, \sigma) \models \mathsf{N} f & \triangleq & (\phi, \sigma_1) \models f
\end{array}
$$

where $\phi \models P$ means the assignment of truth-values given by ϕ satisfies the formula P. The formal definition of $\phi \models P$ is the usual definition for the semantics of propositional formulas.

Defining Sequence

$$
\begin{array}{lcl}
[m \text{ is } 0]^{\phi}\ t\ n & \triangleq & 0 \text{ if } m{=}n \text{ and } t{=}0,\ \text{otherwise } X \\
[m \text{ is } 1]^{\phi}\ t\ n & \triangleq & 1 \text{ if } m{=}n \text{ and } t{=}0,\ \text{otherwise } X \\
[f_1 \text{ and } f_2]^{\phi}\ t\ n & \triangleq & ([f_1]^{\phi}\ t\ n) \sqcup ([f_2]^{\phi}\ t\ n) \\
[f \text{ when } P]^{\phi}\ t\ n & \triangleq & [f]^{\phi}\ t\ n \text{ if } \phi \models P,\ \text{otherwise } X \\
[\mathsf{N}\, f]^{\phi}\ t\ n & \triangleq & [f]^{\phi}\ (t{-}1)\ n \text{ if } t{\neq}0,\ \text{otherwise } X
\end{array}
$$

Defining Trajectory

$$
\begin{aligned}
[\![f]\!]^{\phi}\ \mathcal{M}\ 0\ n &\triangleq [f]^{\phi}\ 0\ n \\
[\![f]\!]^{\phi}\ \mathcal{M}\ t\ n &\triangleq [f]^{\phi}\ t\ n \sqcup \mathcal{M}\ ([\![f]\!]^{\phi}\ \mathcal{M}\ (t-1))\ n
\end{aligned}
$$

STE Implementation

$$\vdash \ \mathcal{M} \models A \Rightarrow C \quad \equiv \quad \forall t\, n.\ [C]^{\phi}\ t\ n \ \sqsubseteq \ [\![A]\!]^{\phi}\ \mathcal{M}\ t\ n$$

Symmetry and STE

Symmetry Theory for STE

Permutation on states

$$apply_s\, \pi\, s \triangleq \lambda n.\ s(\pi\ n)$$

Permutation on sequences

$$apply_\sigma\, \pi\, \sigma \triangleq \lambda t\, n.\ \sigma\ t\ (\pi\ n)$$

Property of swap

$$is\_swap\ \ \pi \triangleq \forall a\, b.\ (\pi(a) = b) \supset (\pi\,(b) = a)$$

Symmetry of STE models

$$Sym_{\chi}\ \mathcal{M}\ \pi \ \triangleq\ \forall s.\ apply_s\, \pi\, (\mathcal{M}\, s) \ =\ \mathcal{M}\, (apply_s\, \pi\, s)$$

Permutation and Sequences

$$\vdash \ \forall \pi.\ \mathit{is\_swap}\ \pi\ \supset$$
$$\forall \sigma_1\ \sigma_2.\ (\sigma_1\ \ddot{\sqsubseteq}\ \sigma_2\ \equiv\ (\mathit{apply}_\sigma\, \pi\, \sigma_1)\ \ddot{\sqsubseteq}\ (\mathit{apply}_\sigma\, \pi\, \sigma_2))$$

Permutation on Trajectory Formulas

$$
\begin{aligned}
\mathit{apply}_f\ \pi\ f \quad \triangleq \quad & (\pi\ n)\ \text{ is }\ 0 \\
& \mid (\pi\ n)\ \text{ is }\ 1 \\
& \mid (\pi\ f)\ \text{ and }\ (\pi\ g) \\
& \mid (\pi\ f)\ \text{ when }\ P \\
& \mid \mathsf{N}\,(\pi\ f)
\end{aligned}
$$

Two Important Lemmas

Defining Sequence Lemma

$$\forall \pi.\ is\_swap\ \ \pi \supset \forall \phi f\, t\, n.\ (apply_\sigma\, \pi\, [f]^\phi\, t\ n\ =\ [apply_f\, \pi\, f]^\phi\, t\ n)$$

Defining Trajectory Lemma

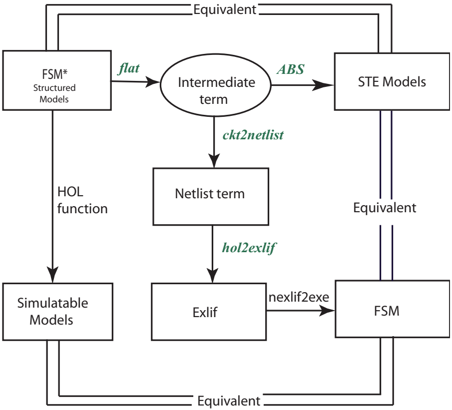
$$\begin{aligned}&\forall \pi.\ is\_swap\ \ \pi \ \supset\\ &\qquad \forall \mathcal{M}.\ Sym_\chi\ \mathcal{M}\, \pi\ \supset\\ &\qquad\qquad \forall \phi f\, t\, n.\ (apply_\sigma\, \pi\, [\![f]\!]^\phi\, \mathcal{M}\ t\ n\ =\ [\![apply_f\, \pi\, f]\!]^\phi\, \mathcal{M}\ t\ n)\end{aligned}$$

Symmetry Soundness Theorem

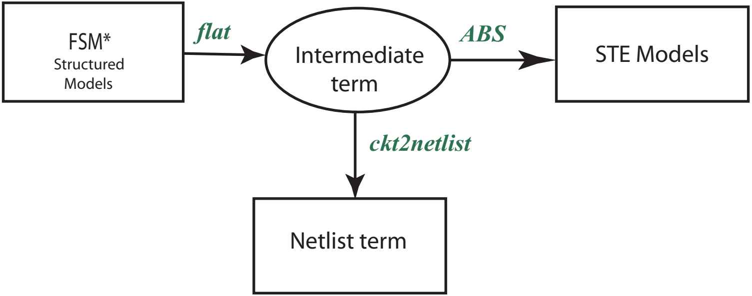
Symmetry Soundness Theorem

$$\forall \vdash \ \mathcal{M}\, \pi\, A\, C.\ \ is\_swap\ \ \pi\ \supset\ Sym_{\chi}\, \mathcal{M}\, \pi$$
$$\supset$$
$$(\mathcal{M}\ \models A\ \Rightarrow\ C\ \equiv\ \mathcal{M}\ \models (apply_f\, \pi\, A)\ \Rightarrow\ (apply_f\, \pi\, C))$$

FSM^* to FSM'



FSM* to FSM$'$


FSM*
Structured
Models
flat
Intermediate
term
ABS
STE Models
ckt2netlist
Netlist term

flat

$$\vdash \ \mathit{flat} \ \ c \ \mathit{nsym} \ \mathit{sym} \ (s_b : \mathit{string} \rightarrow \mathit{bool}) \\ = \ c \ (\mathit{map} \ (\mathit{map} \ s_b) \ \mathit{nsym})(\mathit{map}(\mathit{map} \ s_b) \ \mathit{sym})$$

ckt2netlist

$$\vdash \; ckt2netlist \;\; c \; nsym \; sym \; outp \; (s_b : string \to bool) \; ({s_b}' : string \to bool)$$

$$= \; let \;\; auxflat \;\; c \; nsym \; sym \; outp \; s_b \; {s_b}' = (map(map \; {s_b}') \; outp) \; = \; flat \;\; c \; nsym \; sym \; s_b$$

$$in \;\; auxflat \;\; c \; nsym \; sym \; outp \; s_b \; {s_b}'$$

Drop

Dropping Boolean Values

$$\begin{array}{lrcl} \vdash & drop\ F & = & 0 \\ \wedge & drop\ T & = & 1 \end{array}$$

$$\begin{array}{ll} \vdash & (drop_b\ [\,]\ [\,]\ (s : string \longrightarrow bool \times bool)\ n \ = \ X) \\ \wedge & (drop_b\ [\,]\ \_\ s\ n \ = \ X) \\ \wedge & (drop_b\ \_\ [\,]\ s\ n \ = \ X) \\ \wedge & (drop_b\ ((a : string) :: alist)\ (b :: blist)\ s\ n \ = \\ & \quad (if\ (n = a) \\ & \quad then\ (drop\ b) \\ & \quad else\ drop_b\ alist\ blist\ s\ n)) \end{array}$$

bool2STE

$$
\begin{aligned}
\vdash\ & (\mathit{bool2STE}\ [\,]\ [\,]\ s\ n\ =\ X)\\
\wedge\ & (\mathit{bool2STE}\ [\,]\ \_\ s\ n\ =\ X)\\
\wedge\ & (\mathit{bool2STE}\ \_\ [\,]\ s\ n\ =\ X)\\
\wedge\ & (\mathit{bool2STE}\ (a :: \mathit{alist})\ (b :: \mathit{blist})\ s\ n\ =\\
& \qquad (\mathit{drop}_b\ a\ b\ s\ n)\ \sqcup\ (\mathit{bool2STE}\ \mathit{alist}\ \mathit{blist}\ s\ n))
\end{aligned}
$$

ABS

Generating three-valued models from FSM*

$$
\begin{array}{l}
\vdash \; ABS \;\; c \;\; nsym \;\; sym \;\; outp \;\; (s_b : string \longrightarrow bool) \;\; = \\
\qquad \lambda s : string \longrightarrow bool \times bool. \; \lambda n. \\
\qquad\qquad (let \;\; outp1 \;\; = \;\; flat \;\; c \;\; nsym \;\; sym \;\; s_b \\
\qquad\qquad\qquad in \\
\qquad\qquad (bool2STE \;\; outp \;\; outp1 \;\; s \;\; n))
\end{array}
$$

ABS generates monotonic models

Three valued model is monotonic

$$\vdash \forall c\ nsym\ sym\ outp\ s_b.\ \ Monotonic\ (ABS\ c\ nsym\ sym\ outp\ s_b)$$

Relating swap and π

$$\vdash \ (pi\ (i,j)\ x \ = \ \lambda n.\ if\ (n = el\ i\ x)\ then\ (el\ j\ x)\ else\ if\ (n = el\ j\ x)\ then\ (el\ i\ x)\ else\ n)$$

$$\begin{aligned} \vdash &\ (perm\ (i,j)\ [x] \ = \ pi\ (i,j)\ x) \\ \wedge &\ (perm\ (i,j)\ (x :: xs) \ = \ (pi\ (i,j)\ x) \circ perm\ (i,j)\ xs) \end{aligned}$$

Relating *Sym* and Sym_χ

$$
\begin{array}{l}
\vdash \ \forall c.\ \forall s_b\ nsym.\ Sym\ (c\ (map(map\ s_b)\ nsym)) \ \supset \\
\qquad \forall sym.\ CheckLength\ (map(map\ s_b)\ sym) \ \supset \\
\qquad \forall i\ j.\ \forall outp. \\
\qquad\qquad Sym_\chi\ (ABS\ \ c\ nsym\ sym\ outp\ s_b) \\
\qquad\qquad\qquad (perm\ (i,j)\ (append\ sym\ outp))
\end{array}
$$

Reduction Methodology

Reduction Philosophy

- We present a novel set of inference rules that will help decompose STE to STE′, if used like tactics, and help compose the overall correctness statement STE from STE′, when used in the forward direction.
- Symmetry in circuit models lets us partition the decomposed STE properties into equivalence classes.
- We verify only the representatives and conclude that the other members of the same equivalence classes have been verified as well by way of deduction rather than explicit STE verification.

Inference Rules I

Reflexivity

$$\frac{}{\mathcal{M} \models A \Rightarrow A}$$

Conjunction

$$\frac{\mathcal{M} \models A_1 \Rightarrow B_1 \qquad \mathcal{M} \models A_2 \Rightarrow B_2}{\mathcal{M} \models (A_1 \text{ and } A_2) \Rightarrow (B_1 \text{ and } B_2)}$$

Transitivity

$$\frac{\mathcal{M} \models A \Rightarrow B \qquad \mathcal{M} \models B \Rightarrow C}{\mathcal{M} \models A \Rightarrow C}$$

Inference Rules II

Constraint Implication 1

$$\frac{\mathcal{M} \models A \Rightarrow (C \text{ when } G)}{G \supset (\mathcal{M} \models A \Rightarrow C)}$$

Constraint Implication 2

$$\frac{G \supset (\mathcal{M} \models A \Rightarrow C)}{\mathcal{M} \models A \Rightarrow (C \text{ when } G)}$$

Inference Rules III

Cut

$$\frac{G_1 \supset (\mathcal{M} \models A_1 \Rightarrow B_1) \qquad G_2 \supset (\mathcal{M} \models (B_1 \text{ and } A_2) \Rightarrow C)}{(G_1 \wedge G_2) \supset (\mathcal{M} \models (A_1 \text{ and } A_2) \Rightarrow C)}$$

Specialised Cut

$$\frac{G_1 \supset (\mathcal{M} \models (A \Rightarrow B)) \qquad G_2 \supset (\mathcal{M} \models (B \Rightarrow C))}{(G_1 \wedge G_2) \supset (\mathcal{M} \models (A \Rightarrow C))}$$

Inference Rules IV

Guard Conjunction

$$\frac{G_1 \supset (\mathcal{M} \models A \Rightarrow C) \qquad G_2 \supset (\mathcal{M} \models B \Rightarrow D)}{G_1 \wedge G_2 \supset (\mathcal{M} \models (A \text{ and } B) \Rightarrow C \text{ and } D)}$$

Guard Disjunction

$$\frac{G_1 \supset (\mathcal{M} \models A \Rightarrow C) \qquad G_2 \supset (\mathcal{M} \models B \Rightarrow C)}{G_1 \vee G_2 \supset (\mathcal{M} \models (A \text{ and } B) \Rightarrow C)}$$

Inference Rules V

Antecedent Strengthening 1

$$\frac{\mathcal{M} \models A' \Rightarrow C \qquad [A']^{\phi} \sqsubseteq [A]^{\phi}}{\mathcal{M} \models A \Rightarrow C}$$

Antecedent Strengthening 2

$$\frac{G \;\supset\; (\mathcal{M} \models A' \Rightarrow C) \qquad [A']^{\phi} \sqsubseteq [A]^{\phi}}{G \;\supset\; (\mathcal{M} \models A \Rightarrow C)}$$

Inference Rules VI

Consequent Weakening 1

$$\frac{\mathcal{M} \models A \Rightarrow C' \qquad [C]^{\phi} \sqsubseteq [C']^{\phi}}{\mathcal{M} \models A \Rightarrow C}$$

Consequent Weakening 2

$$\frac{G \supset \mathcal{M} \models A \Rightarrow C' \qquad [C]^{\phi} \sqsubseteq [C']^{\phi}}{G \supset \mathcal{M} \models A \Rightarrow C}$$

Examples and Case Studies

Examples

- Gates - And, Or, Nand, Xor, Xnor etc.
- Comparator
- Mux
- Steering Circuit
- Random Access Memory (RAM)
- Content Addressable Memory (CAM)
- Other circuits with CAMs

Basic gates

Safe functional blocks

$$\begin{array}{llll} \vdash & inv & = & map\ (\sim) \\ \vdash & and & = & fold\ (\wedge)\ id \\ \vdash & or & = & fold\ (\vee)\ id \\ \vdash & nand & = & inv\ \circ\ and \end{array}$$

Basic circuit blocks

$$\begin{array}{llll} \vdash & Inv & = & map\ inv \\ \vdash & And & = & map\ and \\ \vdash & Or & = & map\ or \\ \vdash & Nand & = & map\ nand \end{array}$$

Bitwise operations

$$\begin{aligned} \vdash bAND &= Bitwise\ (\wedge)\ Id \\ \vdash bOR &= Bitwise\ (\vee)\ Id \end{aligned}$$

2-to-1 Multiplexer – FSM*

$$\vdash \quad ctrl\_and\ inp = map\ \ (\wedge\ (hd\ inp))$$
$$\vdash \quad not\_ctrl\_and\ inp = map\ \ (\wedge\ (\sim(hd\ inp)))$$

$$\vdash \quad M1\ inp = (map(ctrl\_and\ inp))\ \circ\ Select\ 0\ Id$$
$$\vdash \quad M2\ inp = (map(not\_ctrl\_and\ inp))\ \circ\ Select\ 1\ Id$$

$$\vdash \quad Auxmux\ inp = ((M1\ inp)\ ||\ (M2\ inp))$$

$$\vdash \quad Mux\ [clk; ctrl] = (map(map\ (DEL\ (hd\ clk)))\ \circ\ Bitwise\ (\vee)\ (Auxmux\ ctrl))$$

2-to-1 Multiplexer – Netlist Term

```
- ckt2netlist Mux [["clk"];["ctrl"]][["a0";"a1"];["b0";"b1"]][["out0";"out1"]] sb sb′

val mux_thm = ⊢ ckt2netlist Mux [["clk"];["ctrl"]]
                                  [["a0";"a1"];["b0";"b1"]]
                                      [["out0";"out1"]] sb sb′ =
(sb′ "out0" = DEL (sb "clk")(∼ sb "ctrl" ∧ sb "b0" ∨ sb "ctrl" ∧ sb "a0"))
∧
(sb′ "out1" = DEL (sb "clk") ( sb "ctrl" ∧ sb "b1" ∨ sb "ctrl" ∧ sb "a1")) : thm
```

2-to-1 Multiplexer – *hol2exlif*

```
- hol2exlif [mux_thm] "mux" "clock"
```

exlif2exe

```
[ashish@clpc1 ashish] nexlif2exe2 mux.exlif
```

Exlif for Mux

```
.model testmux .inputs a0 a1 b0 b1
.outputs out0
.expr n18 = ctrl `
.expr n16 = a0 `
.expr n15 = ctrl `
.expr n13 = b0 `
.expr n12 = n18 '
.expr n10 = n15 & n16
.expr n9 = n12 & n13
.expr n5 = n9 + n10
.expr n4 = clk `
.latch n5 out0 re clock
.inputs a0 a1 b0 b1
.outputs out1
.expr n42 = ctrl `
.expr n40 = a1 `
.expr n39 = ctrl `
.expr n37 = b1 `
.expr n36 = n42 '
.expr n34 = n39 & n40
.expr n33 = n36 & n37
.expr n29 = n33 + n34
.expr n28 = clk `
.latch n29 out1 re clock .end
```

2-to-1 Multiplexer – STE Model

```
``ABS Mux [["clk"];["c"]][["a0";"a1"];["b0";"b1"]] [["out0";"out1"]] s``;

MUX_ABS_CONV it;

- val it = ⊢ ABS Mux [["clk"]; ["c"]] [["a0";"a1"];["b0";"b1"]] [["out0";"out1"]] s
        = (λ s′ n. (if n = "out0" then
                        (if ∼s "c" ∧ s "b0" ∨ s "c" ∧ s "a0"
                                then One else Zero)
        else (if n = "out1" then
                        (if ∼s "c" ∧ s "b1" ∨ s "c" ∧ s "a1"
                                then One else Zero) else X)) lub X) : thm
```

Property verification

$$
\begin{aligned}
Mux \models\ & (''a_0'' \text{ is } a_0) \text{ and } (''a_1'' \text{ is } a_1) \text{ and } (''a_2'' \text{ is } a_2) \\
& \text{and } (''b_0'' \text{ is } b_0) \text{ and } (''b_1'' \text{ is } b_1) \text{ and } (''b_2'' \text{ is } b_2) \\
& \text{and } (''ctrl'' \text{ is } c) \quad \Rightarrow \\
& ((''out_0'' \text{ is } a_0) \text{ and } (''out_1'' \text{ is } a_1) \text{ and } (''out_2'' \text{ is } a_2)) \text{ when } c \\
& \text{and} \\
& ((''out_0'' \text{ is } b_0) \text{ and } (''out_1'' \text{ is } b_1) \text{ and } (''out_2'' \text{ is } b_2)) \text{ when } \bar{c})
\end{aligned}
$$

We shall use STE inference rules to decompose this property into several smaller properties.

Verification in the presence of symmetry – I

Using *Conjunction* on the antecedent and consequent, we get the following goals

$$(1)\ \ Mux \models ("a_0" \text{ is } a_0) \text{ and } ("b_0" \text{ is } b_0) \text{ and } ("ctrl" \text{ is } c) \Rightarrow (("out_0" \text{ is } a_0) \text{ when } c) \text{ and } (("out_0" \text{ is } b_0) \text{ when } \bar{c})$$

$$(2)\ \ Mux \models ("a_1" \text{ is } a_1) \text{ and } ("b_1" \text{ is } b_1) \text{ and } ("ctrl" \text{ is } c) \Rightarrow (("out_1" \text{ is } a_1) \text{ when } c) \text{ and } (("out_1" \text{ is } b_1) \text{ when } \bar{c})$$

$$(3)\ \ Mux \models ("a_2" \text{ is } a_2) \text{ and } ("b_2" \text{ is } b_2) \text{ and } ("ctrl" \text{ is } c) \Rightarrow (("out_2" \text{ is } a_2) \text{ when } c) \text{ and } (("out_2" \text{ is } b_2) \text{ when } \bar{c})$$

Verification in the presence of symmetry – II

We do an STE run to verify (1). Mux exhibits symmetry - exchange the first line with the second, and the first with the third, and Sym_χ *Mux* π holds, therefore by using *Symmetry Soundness Theorem* we can conclude that (2) and (3) are verified as well.

Gist

Thus verifying an n-bit 2-to-1 mux entails verifying a 1-bit mux using only two symbolic variables, and by way of using symmetry arguments, and inference rules, we can conclude that the n-bit mux is verified as well.
In general verifying an $m - to - 1$ Mux with $n - bit$ wide input buses will require m distinct symbolic variables for input buses and $\log m$ variable for selecting one of the m inputs.

Comparator

FSM*

$$\vdash \; xnor\ a\ b = (a \wedge b) \vee (\sim a \wedge \sim b)$$

$$\vdash \; Comp\,[\![ck]\!] = $$
$$let\ \ comp1 = Bitwise\ \ xnor\ \ Id\ \ in$$
$$map(map(DEL\ ck)) \circ And \circ comp1$$

Property Reduction

$$
\begin{aligned}
Comp \quad \models \quad & (''a_0'' \text{ is } a_0) \text{ and } (''b_0'' \text{ is } b_0) \text{ and} \\
& (''a_1'' \text{ is } a_1) \text{ and } (''b_1'' \text{ is } b_1)
\end{aligned}
$$

$$\Rightarrow$$

$$
\begin{aligned}
&(''out'' \text{ is } 1) \text{ when } ((a_0 = b_0) \wedge (a_1 = b_1)) \text{ and} \\
&(''out'' \text{ is } 0) \text{ when } (\sim(a_0 = b_0) \vee (\sim(a_1 = b_1)))
\end{aligned}
$$

Verification in the presence of symmetry – I

Equality

$$
\begin{aligned}
Comp \models\ & (''a_0'' \text{ is } a_0) \text{ and } (''b_0'' \text{ is } b_0) \text{ and} \\
& (''a_1'' \text{ is } a_1) \text{ and } (''b_1'' \text{ is } b_1) \\
& \Rightarrow (''out'' \text{ is } 1) \text{ when } ((a_0 = b_0) \wedge (a_1 = b_1))
\end{aligned}
$$

Inequality

$$
\begin{aligned}
Comp \models\ & (''a_0'' \text{ is } a_0) \text{ and } (''b_0'' \text{ is } b_0) \text{ and} \\
& (''a_1'' \text{ is } a_1) \text{ and } (''b_1'' \text{ is } b_1) \\
& \Rightarrow (''out'' \text{ is } 0) \text{ when } (\sim(a_0 = b_0) \vee (\sim(a_1 = b_1)))
\end{aligned}
$$

Verification in the presence of symmetry – Equality

The goal is to show that

$$
\begin{aligned}
Comp \models\ & (''a_0''\ \text{is}\ a_0)\ \text{and}\ (''b_0''\ \text{is}\ b_0)\ \text{and}\\
& (''a_1''\ \text{is}\ a_1)\ \text{and}\ (''b_1''\ \text{is}\ b_1)\\
& \Rightarrow (''out''\ \text{is}\ 1)\ \text{when}\ ((a_0 = b_0) \wedge (a_1 = b_1))\\
let\ B_0 &= (''I_0''\ \text{is}\ 1)\\
let\ B_1 &= (''I_1''\ \text{is}\ 1)\\
let\ A_0 &= (''a_0''\ \text{is}\ a_0)\ \text{and}\ (''b_0''\ \text{is}\ b_0)\\
let\ A_1 &= (''a_1''\ \text{is}\ a_1)\ \text{and}\ (''b_1''\ \text{is}\ b_1)\\
let\ G_0 &= (a_0 = b_0)\\
let\ G_1 &= (a_1 = b_1)\\
let\ C &= ''out''\ \text{is}\ 1
\end{aligned}
$$

Verification in the presence of symmetry – Equality

$$Comp \models A_0 \Rightarrow (B_0 \text{ when } G_0) \qquad (STE\ \ run)$$

$$Comp \models A_1 \Rightarrow (B_1 \text{ when } G_1) \qquad (Symmetry)$$

$$G_0 \supset (Comp \models A_0 \Rightarrow B_0) \qquad (Constraint\ Implication\ 1)$$

$$G_1 \supset (Comp \models A_1 \Rightarrow B_1) \qquad (Constraint\ Implication\ 1)$$

Verification in the presence of symmetry – Equality

From Guard Conjunction we get

$(G_0 \ \wedge \ G_1) \ \supset \ (Comp \ \models \ (A_0 \text{ and } A_1) \ \Rightarrow (B_0 \text{ and } B_1))$

By STE run

$Comp \ \models \ (B_0 \text{ and } B_1) \Rightarrow C$

By Specialised Cut we get

$(G_0 \wedge \ G_1) \supset \ (Comp \ \models \ A_0 \text{ and } A_1 \Rightarrow C)$

By Constraint Implication 2 *we get*

$Comp \ \models \ (A_0 \text{ and } A_1) \Rightarrow (C \text{ when } (G_0 \wedge G_1))$

Verification in the presence of symmetry – Equality

Replacing the values of A_0, A_1, C, G_0 *and* G_1 *we get*

$$
\begin{aligned}
Comp \models \; & (''a_0'' \text{ is } a_0) \text{ and } (''b_0'' \text{ is } b_0) \text{ and} \\
& (''a_1'' \text{ is } a_1) \text{ and } (''b_1'' \text{ is } b_1) \\
& \Rightarrow (''out'' \text{ is } 1) \text{ when } ((a_0 = b_0) \wedge (a_1 = b_1))
\end{aligned}
$$

Verification in the presence of symmetry – Inequality

$$
\begin{aligned}
Comp \models\ & (''a_0'' \text{ is } a_0) \text{ and } (''b_0'' \text{ is } b_0) \text{ and} \\
& (''a_1'' \text{ is } a_1) \text{ and } (''b_1'' \text{ is } b_1) \\
& \Rightarrow (''out'' \text{ is } 0) \text{ when } (\sim(a_0 = b_0) \vee (\sim(a_1 = b_1)))
\end{aligned}
$$

$$
\begin{aligned}
let\ A = & (''a_0'' \text{ is } a_0) \text{ and } (''b_0'' \text{ is } b_0) \text{ and} \\
& (''a_1'' \text{ is } a_1) \text{ and } (''b_1'' \text{ is } b_1)
\end{aligned}
$$

$$
\begin{aligned}
let\ A_0 &= (''a_0'' \text{ is } a_0) \text{ and } (''b_0'' \text{ is } b_0) \\
let\ A_1 &= (''a_1'' \text{ is } a_1) \text{ and } (''b_1'' \text{ is } b_1) \\
let\ C &= (''out'' \text{ is } 0) \\
let\ G_0 &= \sim(a_0 = b_0) \\
let\ G_1 &= \sim(a_1 = b_1)
\end{aligned}
$$

Verification in the presence of symmetry – Inequality

$$Comp \models A_0 \Rightarrow C \text{ when } G_0 \quad (STE\ run)$$

$$Comp \models A_1 \Rightarrow C \text{ when } G_1 \quad (Symmetry)$$

$$G_0 \supset (Comp \models A_0 \Rightarrow C) \quad (Constraint\ Implication\ 1)$$

$$G_1 \supset (Comp \models A_1 \Rightarrow C) \quad (Constraint\ Implication\ 1)$$

$$G_0 \lor G_1 \supset (Comp \models ((A_0 \text{ and } A_1) \Rightarrow C))(Constraint\ Disjunction)$$

$$Comp \models (A_0 \text{ and } A_1) \Rightarrow C \text{ when } (G_0 \lor G_1) \quad (Constraint\ Implication\ 2)$$

Verification in the presence of symmetry – Inequality

Replacing values we get

$$\begin{aligned} Comp \models \; & (''a_0'' \text{ is } a_0) \text{ and } (''b_0'' \text{ is } b_0) \text{ and} \\ & (''a_1'' \text{ is } a_1) \text{ and } (''b_1'' \text{ is } b_1) \\ & \Rightarrow (''out'' \text{ is } 0) \text{ when } (\sim(a_0 = b_0) \vee (\sim(a_1 = b_1))) \end{aligned}$$

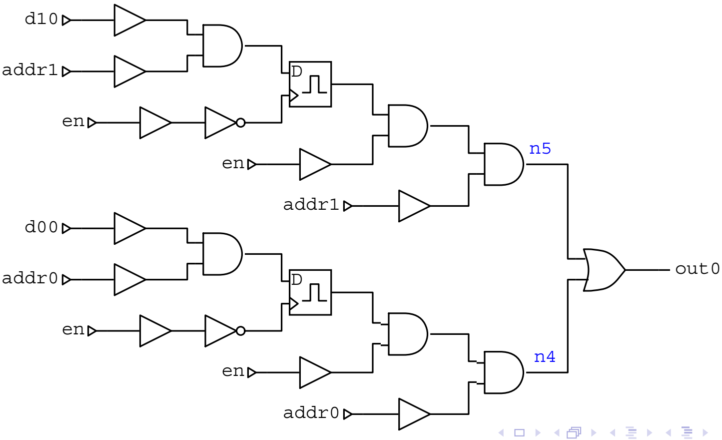
Gist

We can verify an n-bit comparator requires only two variables instead of $2n$. Therefore the BDDs that get built stay really small.

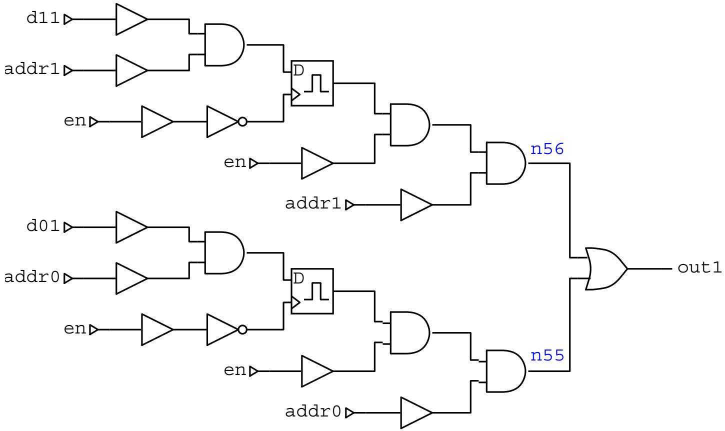
RAM – FSM*

```
⊢ CTRL_AND inp = MAP (∧ (HD inp)) o id

//for a given addr line does the and with all the data bits
⊢ (NBITS [] = NULL)
∧ (NBITS ([]::xs) = NULL)
∧ (NBITS [a::addr_list] =
let n = (LENGTH (a::addr_list) - 1) in
(NBITS [addr_list]) || (MAP (CTRL_AND [a]) o (SELECT n ID)))
∧ (NBITS ((x::y)::xs) = NULL)

//one line of memory --n bits
⊢ oneline [[rw]] [[addr]] =
MAP (CTRL_AND [addr]) o (MAP (CTRL_AND [rw])) o
(MAP (MAP (AH ( rw)))) o NBITS [[addr]]

//generate n lines
⊢ (NLineMem en [[]] = NULL)
∧ (NLineMem en [(x::xs)] =
let n = (LENGTH (x::xs) - 1) in
(((oneline en [[x]]) o (SELECT n ID)) || (NLineMem en [xs])))

// m X n memory
⊢ memory en addr =
(BITWISE ∨ ID) o (NLineMem en addr)
```

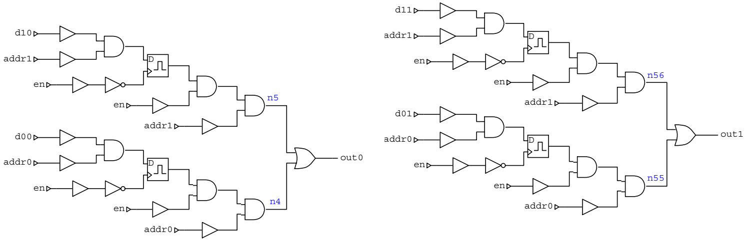
RAM – Memory Lines as seen in Forte I



RAM – Memory Lines as seen in Forte II



RAM – Memory Lines as seen in Forte III



RAM – Property Reduction

```
// symmetry based reduction

// populate the first column with symbolic address and data values
let A0 = (("addr0" is addr0) and ("addr1" is addr1)) from 0 to 5

//populating the first column
let D0 = (("d00" is d00) and ("d10" is d10)) from 0 to 1

// write takes place in the first cycle followed by read enabled
let en = ("en" is F from 0 to 1) and ("en" is T from 1 to 5)

//output of the first column
let B0 = (("n4" is (addr0 ∧ d00)) from 1 to 2) and
         (("n5" is (addr1 ∧ d10)) from 1 to 2)

let trace = map (λn. n,0,5)(nodes memory)
```

RAM – Property Reduction

```
//A0 ⇒ B0 (by STE run)
STE "-s -w" memory [] (A0 and D0 and en) B0 trace

// output of the 0th bit
let C0 = ("out0" is ((addr0 ∧ d00) ∨ (addr1 ∧ d10))) from 1 to 2

//B0 ⇒ C0 (by STE run)
STE "-s -w" memory [] B0 C0 trace

// Specialised Cut
STE "-s -w" memory [] (A0 and D0 and en) C0 trace
```

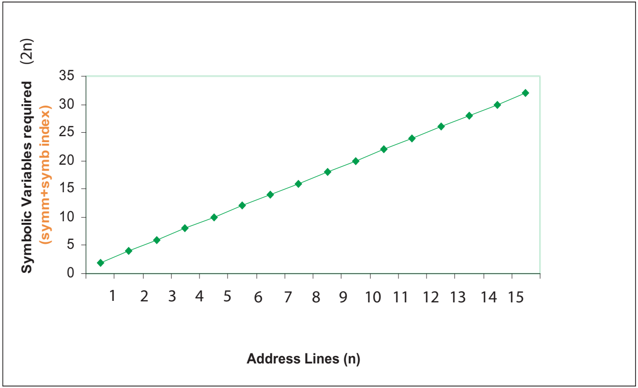
RAM – Property Reduction

```
// Now for the second column,
// of course we never do this but infer from Symmetry

pi = "d00" ⇝ "d01",
     "d10" ⇝ "d11",
     "n5" ⇝ "n56",
     "n4" ⇝ "n55",
     "out0" ⇝ "out1"

let A1 = (("addr0" is addr0) and ("addr1" is addr1)) from 0 to 5
let D1 = (("d01" is d01) and ("d11" is d11)) from 0 to 1
let B1 = (("n55" is (addr0 ∧ d01)) from 1 to 2) and
          (("n56" is (addr1 ∧ d11)) from 1 to 2)

//A1 ⇒ B1 (by STE run)
STE "-s -w" memory [] (A1 and D1 and en) B1 trace

// output of the 1st bit
let C1 = ("out1" is ((addr0 ∧ d01) ∨ (addr1 ∧ d11))) from 1 to 2

//B1 ⇒ C1 (by STE run)
STE "-s -w" memory [] B1 C1 trace
```

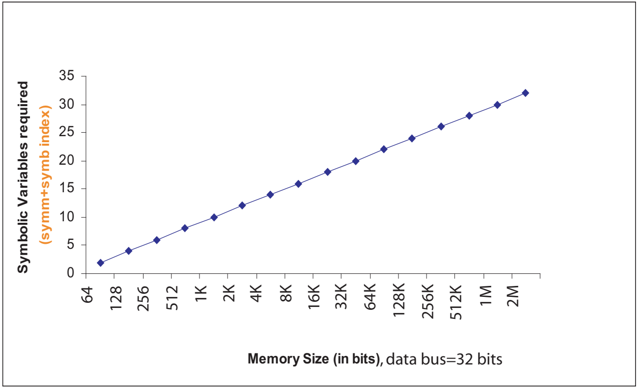
RAM – Property Reduction

```
// Specialised Cut
STE "-s -w" memory [] (A1 and D1 and en) C1 trace

//STE Conjunction
STE "-s -w" memory [] (A0 and A1 and D0 and D1 and en) (C0 and C1) trace

// Antecedent Weakening
STE "-s -w" memory [] (A0 and D0 and D1 and en) (C0 and C1) trace
```

RAM – Our memory requirement I



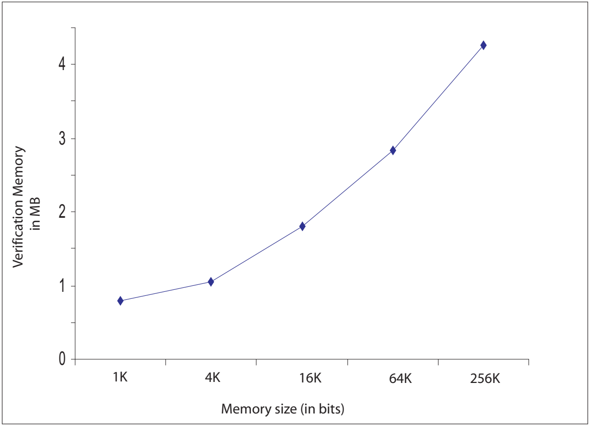
RAM – Our memory requirement II



RAM – Our memory requirement III

width of addr bus	address lines	memory sz (bits) data=32 bits	variables reqd (symmetry + symb indexing)
1	2	64	2
2	4	128	4
3	8	256	6
4	16	512	8
5	32	1k	10
6	64	2k	12
7	128	4k	14
8	256	8k	16
9	512	16k	18
10	1k	32k	20
11	2k	64k	22
12	4k	128k	24
13	8k	256k	26
14	16k	512k	28
15	32k	1M	30
16	64k	2M	32

Pandey's RAM Verification



Pandey's RAM Verification versus Us

Pandey's method

- Even though the verification memory requirment seems to scale nearly linearly.
- Substantial time and memory is used in isomorphism checks.
- Computing reduced models by using symmetry, costing substantial extra time and memory (see page 76-78 of Pandey's thesis).
- Heuristics employed for symmetry detection in SRAM may not be useful for symmetry detection for other circuits for example a CAM.

Our method

- Our requirement for symbolic variables is independent of the size of data bits, it only depends on the number of address lines.
- Our type checking is independent of the size of the RAM, and the type checking takes about a second.
- Type checking and a structured ADT gives us a general method of circuit design and verification.

CAM – FSM*

```
// tag comparison unit
⊢ (tcomparator 0 [[en]] = NULL)
∧ (tcomparator (SUC n) [[en]] =
let intags = (SELECT 0 ID) in
let storedtags = ((MAP(CTRL_AND [en])) o (MAP(MAP(AH ∼en))) o (SELECT (SUC n) ID)) in
                 (mapand o comp1 o (intags || storedtags)) || (tcomparator n [[en]]))

// hit logic
⊢ hit n nsym = BITWISE (∨) (tcomparator n nsym)

⊢ (NBITS [] = NULL)
∧ (NBITS (a::addr_list) =
let n = (LENGTH (a::addr_list) - 1) in
(NBITS addr_list) || (MAP (CTRL_AND a) o (SELECT n ID)))
```

CAM – FSM*

```
// Full CAM
⊢ cam nsym =
let tagen = (HD o HD)((SELECT 0 ID) nsym) in
let dataen = (HD o HD)((SELECT 1 ID) nsym) in
let n = LENGTH (ID nsym) - 3 in
let match = tcomparator n [[tagen]]
              (((TAIL o TAIL)ID) nsym) in
let data = (NBITS match) o (MAP(CTRL_AND [dataen])) o
                   (MAP(MAP(AH(~dataen)))) o ID
                    in (BITWISE (∨) data)
```

CAM – Towards the Netlist

```
val cam_thm = ⊢ ckt2netlist cam [["tagen"];["dataen"];
                    ["Tag[0]";"Tag[1]"];
                    ["t0[0]";"t0[1]"];
                    ["t1[0]";"t1[1]"]]
                   [["d0[0]";"d0[1]"];
                    ["d1[0]";"d1[1]"]]
                   [["out[0]";"out[1]"]] sb sb'

val hit_thm = ⊢ ckt2netlist (hit 2) [["tagen"]]
               [["Tag[0]";"Tag[1]"];
                ["t0[0]";"t0[1]"];
                ["t1[0]";"t1[1]"]] [["hit"]] sb sb'
```

hol2exlif

```
- hol2exlif [cam_thm, hit_thm] "cam" "";
```

exlif2exe

```
[ashish@clpc1 ashish] nexlif2exe2 cam.exlif
```

Property Reduction – Initialising values

```
//initialising variables
//data stored in both the lines
let d00 = variable "d0[0]";
let d01 = variable "d0[1]";
let d10 = variable "d1[0]";
let d11 = variable "d1[1]";

//tags stored in the lines
let t00 = variable "t0[0]";
let t01 = variable "t0[1]";
let t10 = variable "t1[0]";
let t11 = variable "t1[1]";

//input tags
let Tag0 = variable "Tag[0]";
let Tag1 = variable "Tag[1]";

//read enabled and incoming tag takes on symbolic values
let base_ant = ((("Tag[0]" is Tag0) and ("Tag[1]" is Tag1)) from 0 to 2)
                and ("tagen" is F from 0 to 1) and ("tagen" is T from 1 to 2)
                 and ("dataen" is F from 0 to 1) and ("dataen" is T from 1 to 2);;
```

Property Reduction – Initialising values

```
//populate the tags in the first line
let A0_0 = ((("T0[0]" is t10) and ("T0[1]" is t11)) from 0 to 1) and base_ant;

//populate the data in the first line
let A0_1 = ((("d0[0]" is d10) and ("d0[1]" is d11)) from 0 to 1) and base_ant;

//populate the tags in the second line
let A1_0 = ((("T1[0]" is t10) and ("T1[1]" is t11)) from 0 to 1) and base_ant;

//populate the data in the second line
let A1_1 = ((("d1[0]" is d10) and ("d1[1]" is d11)) from 0 to 1) and base_ant;

let A0 = A0_0 and A0_1;
let A1 = A1_0 and A1_1;

//data stored at the first line appears at the output
let C0 = (("out[0]" is d00) and ("out[1]" is d01)) from 1 to 2;

//data stored at the second line appears at the output
let C1 = (("out[0]" is d10) and ("out[1]" is d11)) from 1 to 2;
```

Property Reduction – Initialising values

```
//incoming tags match the tags stored at the first line
let G0 = (Tag0 = t00) ∧ (Tag1 = t01);

//incoming tags match the tags stored at the second line
let G1 = (Tag0 = t10) ∧ (Tag1 = t11);

//incoming tags don't match the tags stored at the first line
let nG0 = NOT G0;

//incoming tags don't match the tags stored at the second line
let nG1 = NOT G1;

//hit[0] is 0
let B0_0 = "hit[0]" is F from 1 to 2;

//hit[0] is 1
let B0_1 = "hit[0]" is T from 1 to 2;

//hit[1] is 0
let B1_0 = "hit[1]" is F from 1 to 2;

//hit[1] is 1
let B1_1 = "hit[1]" is T from 1 to 2;
```

Property Reduction – CAM read

Correct Data is read I

```
let trace = map (λn.n,0,2) (nodes cam_fsm);

//By STE run, using the comparator verification strategy as in inequality case
using only two variables for tag comparison
nG0 ⊃ (STE "-s -w" cam_fsm [] A0_0 B0_0 trace);

//Using Antecedent Strengthening
nG0 ⊃ (STE "-s -w" cam_fsm [] (A0_0 and A0_1) B0_0 trace);

//But (A0_0 and A0_1) = A0, so we have
nG0 ⊃ (STE "-s -w" cam_fsm A0 B0_0) (1)

//Now we shall show how to deduce the correctness property
G1 ⊃ (STE "-s -w" cam_fsm [] (B0_0 and A1) C1 trace);
```

Property Reduction – CAM read

Correct Data is read I

```
//comparator verification strategy,
//using only variables for the tag of the second line
G1 ⊃ (STE "-s -w" cam [] A1_0 B1_1 trace); (2)

// STE run using only one data variable, and using symmetry of the data bus to deduce
(STE "-s -w" cam [] (B1_1 and (A1_1 and B0_0)) C1 trace); (3)

//Using Cut on (2) and (3) we get
G1 ⊃ (STE "-s -w" cam [] (B0_0 and A1_0 and A1_1) C1 trace); (4)

//But A1_0 and A1_1 = A1, therefore
G1 ⊃ (STE "-s -w" cam [] (B0_0 and A1) C1 trace); (5)

//By Guard Conjunction and the Cut Rule on (1) and (5), we can deduce
(nG0 ∧ G1) ⊃ (STE "-s -w" cam_fsm [] (A0 and A1) C1 trace);

//By Constraint Implication 2, we can deduce
(STE "-s -w" cam_fsm [] (A0 and A1)(C1 when (nG0 ∧ G1)) trace);
```

Property Reduction – CAM read

Correct Data is read II

```
//By repeating the same strategy for the second CAM line
(STE "-s -w" cam_fsm [] (A0 and A1)(C0 when (nG1 ∧ G0)) trace);
```

Property Reduction - Correct Data Read

Overall correctness assertion

```
//By STE Conjunction
(STE "-s -w" cam_fsm [] (A0 and A1) ((C0 when (nG1 ∧ G0)) and
                                     (C1 when (nG0 ∧ G1))) trace);
```

Property Reduction – Hit Logic

Hit rises if there is a match

```
//hit is 1
let C = "hit" is T from 1 to 2;


//hit is 1 if the tags match at the first line
// STE run uses only two variables, comparator reduction strategy
G0 ⊃ (STE "-s -w" cam_fsm [] A0 C trace);


//hit is 1 if the tags match at the second line
// STE run uses only two variables, comparator reduction strategy
G1 ⊃ (STE "-s -w" cam_fsm [] A1 C trace);


//By Guard Disjunction we conclude
(G0 ∨ G1) ⊃ (STE "-s -w" cam_fsm [] (A0 and A1) C trace);
```

Hit stays low of there is no match

```
//hit[0] is 0
let hit0 = "hit0" is F from 1 to 2;

//hit[1] is 0
let hit1 = "hit1" is F from 1 to 2;

//hit is 0
let C = "hit" is F from 0 to 2;

//By STE run using only two variables, comparator verification strategy
nG0 ⊃ (STE "-s -w" cam_fsm [] A0 hit0 trace);

//By STE run using only two variables, comparator verification strategy
nG1 ⊃ (STE "-s -w" cam_fsm [] A1 hit1 trace);

//By Guard Conjunction
(nG0 ∧ nG1) ⊃ (STE"-s -w" cam_fsm [](A0 and A1) (hit0 and hit1) trace);

//By STE run
(STE "-s -w" cam_fsm [] (hit0 and hit1) C trace);

//Applying the Specialised Cut we conclude
(nG0 ∧ nG1) ⊃ (STE "-s -w" cam_fsm [] (A0 and A1) C trace);
```

Our memory and time requirement

Gist - Correct Data Read

For a CAM with n lines and tag width t and data width d, we need to use only *two* variables at any one time for tag comparison and *one* variable for data bit to verify the correct data read property. The space complexity is reduced from $n * (t + d) + t$ to 3.
The time complexity is linear with respect to the number of CAM lines.

Gist - Hit Logic

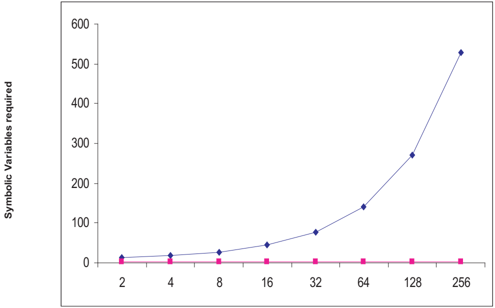
For verifying the hit logic, we need only two variables at any point of time, for any number of CAM lines, tag entries and data entries! The time complexity is linear with respect to the number of CAM lines.

Pandey's CAM verification

Pandey's CAM verification

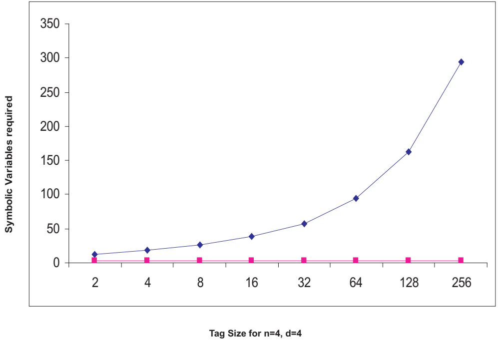
- Pandey's CAM encoding requires $\log_2 n + n * \log_2 t + t + d$ variables for verification of data read and hit logic. Symmetry is not used at all, only symbolic indexing used.
- For a 64 line CAM with 32 bit tags and 32 bit data, he would need 6+(64*5)+32+32=390 variables whereas we would need 3 for correct data read property and 2 for the hit logic.

CAM – BDD Variables Required wrt CAM size

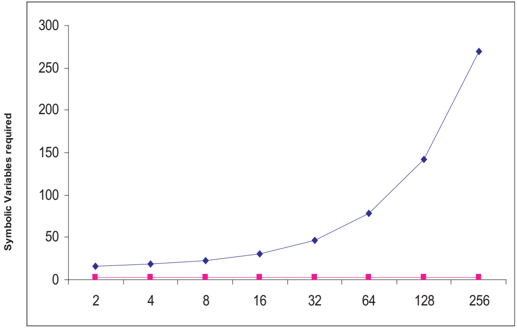


Number of CAM lines for t=4, d=4

CAM – BDD Variables Required wrt tag size



CAM – BDD Variables Required wrt data size



Data size for n=4, t=4

Related Work

Symmetry in Model Checking

- Pandey and Bryant – Verification of memory arrays
- Ip and Dill, Ken McMillan – Scalarsets in Murphi and SMV
- Sistla, Emerson and Jha – Symmetry and model checking
- Sistla – Symmetry based model checker
- Bill Roscoe, Ranko Lazic, Tom Newcomb – Data independence

Designing structured models

- Mary Sheeran, Wayne Luk – Ruby
- Mary Sheeran, Satnam Singh et.al. – Lava
- O' Donnell – Netlist generator from functional language
- Chavan, Woo Min and Shiu-Kai Chin – HOL2GDT – Desiging a mulitplier chip from specifications in HOL
- Tom Melham – Mini-Lava in reFLect

Conclusions and Future Work

- Dealing with other kinds of structural symmetry – perhaps more richer type of structured models is needed.
- Data symmetry and temporal symmetry.
- Feedback is not implemented at present.
- Lists are not the most appropriate data structure.