6. Symbolic Trajectory Evaluation, Term Rewriting

Jacob Abraham

Department of Electrical and Computer Engineering
The University of Texas at Austin

Verification of Digital Systems
Spring 2020

February 6, 2020

Motivation for Symbolic Trajectory Evaluation

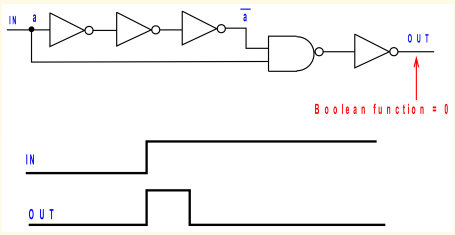
- Equivalence checking between RTL and circuit schematics is difficult for some circuits (e.g., custom arrays)
  - Critical timing and self-timed control logic
  - Large number of bit-cells
  - Inherently complex sequential logic blocks
  - Dynamic logic
- Traditional tools fail on such circuits
  - Very large state space, too many initial state/input sequences for simulation-based tools
  - Boolean equivalence tools only check static cones of logic, do not capture dynamic behavior

Acknowledgements: J. Bhadra, J. Harrison, K. Claessen, J.-W. Roorda, N. Krishnamurthy, A. K. Martin, H. Anand, J. Yang, F. Xie

ECE Department, University of Texas at Austin Rewriting Jacob Abraham, February 6, 2020 1 / 1

Motivation

- Control for customer array structure



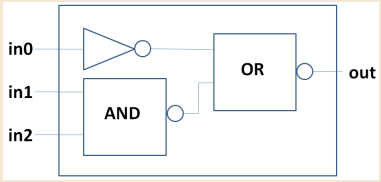
- With zero delay functional simulation OUT is 0
- OUT pulse fans out to array READ/WRITE control signals
- Need to define unit delay for all gates to get a pulse on OUT
- Verification for correctness fails in downstream logic without this pulse

Symbolic Trajectory Evaluation

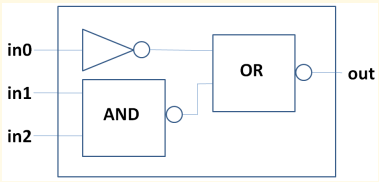
- Symbolic Trajectory Evaluation (STE) is a high-performance simulation-based model checking technique, originally invented by Seger and Bryant
- STE uses a combination of *three-valued simulation and symbolic simulation*

Example to illustrate STE

- The specification says that out represents the output of a 3-input AND gate with in0, in1, and in2 as inputs
- An implementation of the above spec



Scalar Simulation



- We need 2^n (8 here) simulation patterns
- One such pattern is the assertion
  - (in0 **is** 0) **and** (in1 **is** 1) **and** (in2 **is** 0) $\implies$ (out **is** 0)
  - Every check like this is of the form $A \implies C$, where A is an *antecedent* and C is a *consequent*
  - All STE assertions are of this form – we will give formal definition of STE assertions and define their semantics
- Not practical for large scale designs and so we need to make it more efficient, but how?

Three-valued Simulation

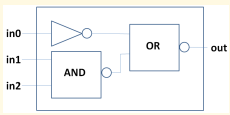
- Observation
  - Any input having assigned to 0, makes the output 0
  - Other inputs do not matter meaning could be 0 or 1 (X)
- If we introduce a new value (X = don't care) in the simulation we need to evaluate the outputs of standard gates

x	$\neg x$
0	1
1	0
X	X

x	y	$x.y$
0	0	0
0	1	0
1	0	0
1	1	1
X	0	0
0	X	0
X	1	X
1	X	X
X	X	X

x	y	$x+y$
0	0	0
0	1	1
1	0	1
1	1	1
X	0	X
0	X	X
X	1	1
1	X	1
X	X	X

Three-valued Simulation

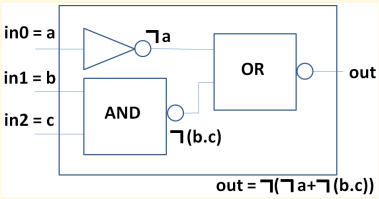


- Cannot afford 2^n simulation patterns and so we will use 3-valued simulations
- Simulation pattern becomes
  - (in0 **is** 0) **and** (in1 **is** X) **and** (in2 **is** X) $\implies$ (out **is** 0)
  - This can be simplified into (in0 **is** 0) $\implies$ (out **is** 0)
  - Still of the form $A \implies C$
- Now the total number of simulations needed has reduced to $n+1$ (4 here)
  - (in1 **is** 0) $\implies$ (out **is** 0)
  - (in2 **is** 0) $\implies$ (out **is** 0)
  - (in0 **is** 1) **and** (in1 **is** 1) **and** (in2 **is** 1) $\implies$ (out **is** 1)
- Can we do better?

Symbolic Simulation

- Assign symbols on inputs
- Now we need to do only *one simulation*
  - (in0 **is** a) **and** (in1 **is** b) **and** (in2 **is** c) $\implies$ (out **is** $a.b.c$)
- Can we do better than 1 simulation?
- No, but...

Checking for correctness



- Boolean expressions can be represented as BDDs (or other data structures) for performing constant time checks
- Verification steps to check the following assertion
  - (in0 **is** a) **and** (in1 **is** b) **and** (in2 **is** c) $\implies$ (out **is** $a.b.c$)
  - Implementation: From the antecedent assign symbols (equivalent to BDDs) to the inputs of the circuit
  - Calculate the symbolic expressions (as BDDs) in the circuit nodes until the symbolic value of the output is known
  - Specification: Calculate the consequent expression BDD
  - Compare the two BDDs for equivalence
- Can we reduce the complexity of this *one simulation*?

Three-valued symbolic simulation

- In STE we combine the efficiency of three-valued simulation with the preciseness of symbolic simulations
- Use this combination by expressing the four 3-valued simulation runs as only one 3-valued symbolic simulation run
- Each assignment of 0 and 1 to two variables represents one 3-valued simulation run and since there are 4 possible assignments, and 4 3-valued simulation runs we can achieve this reduction (symbolic encoding)
- Symbolically,
  $((\neg p.\neg q) \rightarrow$ in0 **is** 0) **and**
  $((\neg p.q) \rightarrow$ in1 **is** 0) **and**
  $((p.\neg q) \rightarrow$ in2 **is** 0) **and**
  $((p.q) \rightarrow$ (in0 **is** 1) **and** (in1 **is** 1) **and** (in2 **is** 1))
  $\implies$ (out **is** $(p.q)$)

Three-valued symbolic simulation

- Assertion has the symbolic form $P \rightarrow A \implies C$ where P is a Boolean expression (predicate)
- Logically, this is an implication
- Simulation-wise this assertion is going to assign values of nodes from A in situations where P is true otherwise, the nodes are kept at Xs

Dual-rail encoding

- The value of a 3-valued variable x can be represented by 2 Boolean variables $x = (x_0, x_1)$

x	(x_0, x_1)
0	(1,0)
1	(0,1)
X	(0,0)

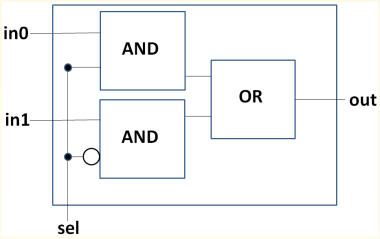
- Operators can be defined as follows
  - NEG: $\neg x = \neg(x_0, x_1) = (x_1, x_0)$
- For 2 3-valued variables $x = (x_0, x_1)$ and $y = (y_0, y_1)$ we can calculate
  - AND: $x.y = (x_0, x_1).(y_0, y_1) = (x_0 + y_0), (x_1.y_1)$
  - OR: $x + y = (x_0, x_1) + (y_0, y_1) = (x_0.y_0), (x_1 + y_1)$

Dual-rail encoding for circuit inputs

- So, this expression implies a few values for in0, in1, in2
  $((\neg p.\neg q) \rightarrow$ in0 **is** 0) **and**
  $((\neg p.q) \rightarrow$ in1 **is** 0) **and**
  $((p.\neg q) \rightarrow$ in2 **is** 0) **and**
  $((p.q) \rightarrow$ (in0 **is** 1) **and** (in1 **is** 1) **and** (in2 **is** 1))
  $\implies$ (out **is** $(p.q)$)
- if $(\neg p.\neg q)$ then assign in0 to 0
  else (if $(p.q)$ then assign in0 1 else keep in0 at an X)
- similar for in1 and in2
- Since in0 is a 3-valued variable its dual-rail encoding will be
  in0 $= ((\neg p.\neg q), (p.q))$
- So, now the verification can be done by *one simulation run*
  with $\lceil log_2(n) \rceil$ (2 here) number of variables

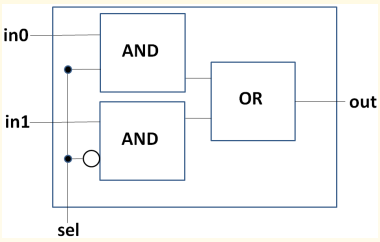
Inaccuracy

We have come from 2^n simulations to one simulation using $\lceil log_2(n) \rceil$ variables so what is the catch?



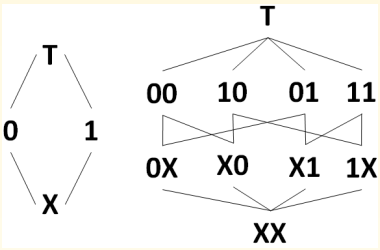
The STE assertion is
(in0 **is** a) **and** (in1 **is** a) $\implies$ (out **is** a)
Without specifying anything about sel in the antecedent it will be kept as an X making out X

Inaccuracy can be reduced by variables



The STE assertion that will resolve the issue is
(sel **is** b) **and**
(in0 **is** a) **and** (in1 **is** a) $\implies$ (out **is** a)
Bottom line: there is a reduction in number of variables but under some circumstances one will need more variables for accuracy

STE Theory: information ordering



- X is "unknown (under-constrained) value of 0 or 1"
- T is "conflicting (over-constrained) value 0 and 1"
- The information ordering forms a lattice that can extend to n variables (X $\preceq$ 0, X $\preceq$ 1, 0 $\preceq \top$ and 1 $\preceq \top$)
- Simulators need to use values $V = \{0, 1, \mathsf{X}, \mathsf{T}\}$
- Simulators use $\neg\top = \top$, $x.\top = \top$, $\top.x = \top$, $x + \top = \top$ and $\top + x = \top$
- Logic gates are monotonic w.r.t. information ordering

Circuit model

- Set of circuit nodes is N (example, in0, in1, out, the inputs and output of an AND gate)
- A state is an assignment of values from V to circuit nodes, $s : N \rightarrow V$ (example, assignment $s(\text{in0}) = \text{X}$, $s(\text{in1}) = 1$, $s(\text{out}) = \text{X}$)
- Circuit state is a collection of such values of the circuit nodes $S = \langle \forall n \in N : s(n) \rangle$ (example, $\langle$ X1X $\rangle$)
  - Note that in STE circuit state includes all nodes and not just latch nodes
- Closure function $F : S \times S$ (example, can be derived from the AND function extended to V)
  - Note that the closure function is not the same as the traditional next state function
  - F propagates given values to other nodes
  - F can be easily constructed from the netlist logic

Trajectory Evaluation Logic (TEL)

STE assertions are of the form $A \implies C$, where A and C are Trajectory Formulas in the language of TEL with the following syntax

Definition

A,C ::= n **is** 0
| n **is** 1
| $A1$ **and** $A2$
| $P \rightarrow A$
| **N** A

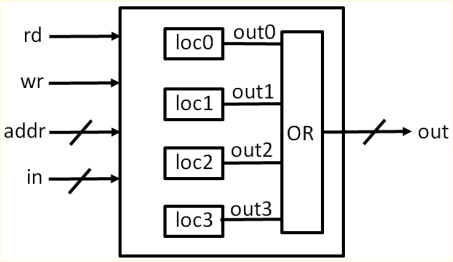
Notes

- P is a predicate over a set of symbolic variables V that are time-independent
- The notion of time is in the form of the next time operator
- We can assign symbolic expressions to node values because (n **is** P) is short form of ($P \rightarrow n$ **is** 1) **and** ($\neg P \rightarrow n$ **is** 0)

A Simplification and the Fundamental Theorem of STE

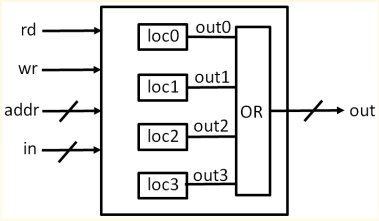
- The simulator performs one simulator run
- The simulator cannot check if C holds for all trajectories
- It calculates the *weakest* trajectory σ in which A is satisfied and checks if C holds on that particular trajectory
- By previous observation C is going to hold on all trajectories that are stronger than σ
- **This implies that it is enough to check C only for the weakest trajectory satisfying A instead of all trajectories**

STE Example: Memory verification



- Memory with address width k and data width n needs $n.2^k$ state holding elements (state based model checkers)
- STE would need $n + k$ variables: let us see how

STE Example: Memory verification

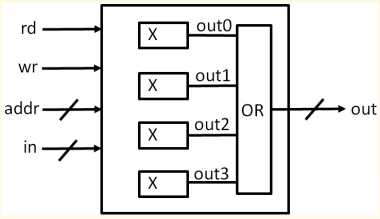


Assertion for a memory
(wr **is** 1) **and** (addr[0] **is** $a0$) **and** (addr[1] **is** $a1$) **and** (in **is** d)
and N ((rd **is** 1) **and** (addr[0] **is** $a0$) **and** (addr[1] **is** $a1$))
$\Longrightarrow$ **N** (out **is** d)
Observations

- Symbolic variables are $a0, a1, d$ (total number is $n + k$)
- Assertion starts from all Xs, writes into symbolic location $a0a1$ with symbolic data d and then reads from the same symbolic location expecting to read out the symbolic data d

STE Example: Memory verification: time 0 and 1

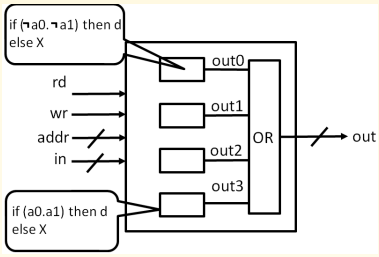
Time 0: before the simulation begins all locations are Xs



Time 1: once written the symbolic values in the memory locations are

- loc0 // ("if I am addressed then I become d else I keep at X")
  $=$ if $(\neg a0.\neg a1 ==$ true) then new loc0 value is d else X
  $=$ if $(\neg a0.\neg a1)$ then d else X
- loc1 $=$ if $(\neg a0.a1)$ then d else X
- loc2 $=$ if $(a0.\neg a1)$ then d else X
- loc3 $=$ if $(a0.a1)$ then d else X

STE Example: Memory verification: time 2



Time 2: outputs are assigned depending on which location is read

- out0 // ("if I am addressed then value of loc0 else 0")
  - $=$ if $(\neg a0.\neg a1)$ then loc0 else 0
  - $=$ if $(\neg a0.\neg a1)$ then (if$(\neg a0.\neg a1)$ then d else X) else 0
  - $=$ if $(\neg a0.\neg a1)$ then d else 0
- Similarly, out1 $=$ if $(\neg a0.a1)$ then d else 0
- out2 $=$ if $(a0.\neg a1)$ then d else 0
- out3 $=$ if $(a0.a1)$ then d else 0

Simplifying, out $=$ out0 OR out1 OR out2 OR out3 $= d$

Symbolic Trajectory Evaluation at Freescale

- VERSYS symbolic trajectory evaluation tool developed at Motorola/Freescale
  - Based on VOSS (from CMU/UBC)
- Trajectory formulas
  - Boolean expressions with the temporal next-time operator
  - Ternary values states represented by a Boolean encoding
- Properties of type: Antecedent $\Longrightarrow$ Consequent
  - Antecedent, Consequent are trajectory formulas
  - Antecedent sets up stimulus, state of the circuit
  - Consequent specifies constraint on the state sequence
- Used to verify PowerPC arrays at Motorola/Freescale in 8 – 10% of the design time
- Bugs found during array equivalence checking
  - Incorrect clock regenerators feeding latches
  - Control logic errors in READ/WRITE enables
  - Violation of "one-hot" property assumptions
  - Scan chain hookup errors
  - Potential circuit-related problems such as glitches and races

Other issues with STE

- LTL with finite number of next time **N** operators
- No notion of initial states
- No concept of reachable states

Term Rewriting Systems

- How a term can be rewritten/transformed into another
- Term rewriting for equivalence checking
- SMT solvers and application to verification and test
- Using term rewriting systems to design and verify processors

Term Rewriting Systems – Greatest Common Divisor

Euclid's Algorithm

- Terms are functions of integers
- Four rules

Rules:

Rule R_1: Gcd(a, b) if b$\neq$0 $\Longrightarrow$ Gcd(b, Rem(a, b))
Rule R_2: Gcd(a, 0) $\Longrightarrow$ a
Rule R_3: Rem(a, b) if $a < b$ $\Longrightarrow$ a
Rule R_4: Rem(a, b) if $a \geq b$ $\Longrightarrow$ Rem(a-b, b)

Example:

Gcd(2,4) $\overset{R_1}{\Longrightarrow}$ Gcd(4, Rem(2,4)) $\overset{R_3}{\Longrightarrow}$ Gcd(4,2)
$\overset{R_1}{\Longrightarrow}$ Gcd(2, Rem(4,2)) $\overset{R_4}{\Longrightarrow}$ Gcd(2, Rem(2,2))
$\overset{R_4}{\Longrightarrow}$ Gcd(2, Rem(0,2)) $\overset{R_3}{\Longrightarrow}$ Gcd(2,0) $\overset{R_1}{\Longrightarrow}$ 2

ECE Department, University of Texas at Austin Rewriting Jacob Abraham, February 6, 2020 26 / 1

Simple Arithmetic Rewriting

Terms

- integer, variable, (,), +, *
- (Note: no evaluation rules defined)

Rules

- Rule1: $(op\ a\ b) \rightarrow (op\ b\ a)$ if $(b < a)$ and $op \in \{+, *\}$
- Rule2: $(*(+\ a\ b)\ c) \rightarrow (+(*\ a\ c)\ (*\ b\ c))$
- Rule3: $(+\ a\ a) \rightarrow (*\ a\ 2)$
- Rule4: $(op\ (op\ a\ b)\ c) \rightarrow (op\ (op\ a\ c)\ b)$ if $(a < c\ \&\ c < b)$ and $op \in \{+, *\}$
- Rule5: $(op\ a\ (op\ b\ c)) \rightarrow (op\ (op\ b\ c)\ a)$ if $(a > c\ \&\ b > c)$ and $op \in \{+, *\}$

Source: Shaun Feng

ECE Department, University of Texas at Austin Rewriting Jacob Abraham, February 6, 2020 27 / 1

Example(s) in Rewriting

Example: $(* \ 4 \ (+ \ 3 \ 3) \stackrel{?}{=} (*(+ \ 4 \ 4) \ 3)$

- $(* \ 4 \ (+ \ 3 \ 3) \rightarrow (*4(* \ 3 \ 2)) \rightarrow (* \ 4 \ (* \ 2 \ 3)) \rightarrow (*(* \ 2 \ 3) \ 4)$
- $(*(+ \ 4 \ 4) \ 3) \rightarrow (* \ (* \ 4 \ 2) \ 3) \rightarrow (* \ (* \ 2 \ 4) \ 3) \rightarrow (*(* \ 2 \ 3) \ 4)$

Prove if $(* \ x(+ \ y \ y) \stackrel{?}{=} (*(+ \ x \ x) \ y)$

Term Rewriting Systems

- 3-tuple: (T, L, R)
  - T: Set of terms (functions, constants, variables, operators) $(t_1, t_2, \ldots, t_n)$
  - L: Set of labels (R1, R2, ...)
  - R: Set of labeled rules (may be conditional) $(r_1, r_2, \ldots, r_n)$
- Rewrite process

  $t_1 \stackrel{r_i}{\rightarrow} t_2 \stackrel{r_j}{\rightarrow} t_3 \stackrel{r_k}{\rightarrow} \ldots \stackrel{r_m}{\rightarrow} t_n$ (Normal Form)
  - Term that cannot be rewritten any further
  - Depending on the system, several normal forms (or no normal form) may exist
  - Normal forms can be used for verification

Equivalence of two terms

- Determine whether the two terms have the same normal forms
- Undecidable in general

Rewriting 3NAND using 2NAND

Terms

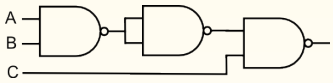
2NAND(), $A, B, C, a, b, \wedge, \neg$

Rules

- Rule $R1$: $a \wedge b \rightarrow \neg$2NAND$(a, b)$
- Rule $R2$: $\neg(\neg a) \rightarrow a$
- Rule $R3$: $\neg a \rightarrow$ 2NAND(a, a)

Apply the rules to get a 3NAND

$\neg((A \wedge B) \wedge C) \xrightarrow{R1} \neg(\neg$2NAND$(A \wedge B, C)) \xrightarrow{R2}$
2NAND$(A \wedge B, C) \xrightarrow{R1}$ 2NAND$(\neg$2NAND$(A, B), C) \xrightarrow{R3}$
2NAND(2NAND(2NAND(A, B), 2NAND(A, B)), C)



Source: Shaun Feng

Termination and Cofluence

Termination

- No infinite rewriting sequence → normal form exists

Cofluence

- Terms can be rewritten in multiple ways, but will eventually yield the same results
  - (*(+ 2 1) (+ 3 4)) → (* 3 (+ 3 4)) → (* 3 7)
  - (*(+ 2 1) (+ 3 4)) → (* (+ 2 1) 7) → (* 3 7)
- Normal form is unique if it exists

Convergence: Termination and Cofluence

- Normal form exists and is unique
- Convergent TRS used in equivalence checking

Rules of TRS Deduction

- (I) Reflexivity: $\overline{t \rightarrow t}$
- (R) Replacement:
  - R1: a $\rightarrow$ a - 3 if a $\in$ I and a $\geq$ 3

$$\frac{t_k(x) \rightarrow t_n(x)}{t_k(x_0/x) \rightarrow t_n(x_0/x)}$$

- (C) Congruence
  - R1: $a \rightarrow$ a - 3 if a $\geq$ 3

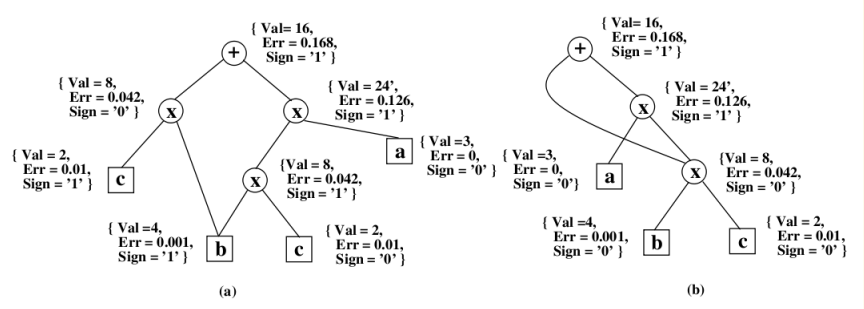
$$\frac{t_1 \rightarrow t'_1, \ldots, t_k \rightarrow t'_m}{f(t_1, \ldots, t_k) \rightarrow f(t'_1, \ldots, t'_k)}$$

- (T) Transitivity

$$\frac{t_1 \rightarrow t_2, \ldots, t_2 \rightarrow t_3}{t_1 \rightarrow t_3}$$

Checking Datapaths Using Arithmetic Expressions

Zhou, 1995
Based on Attribute Syntax Trees
Example: – (a * b * c) + b * c



(a) non-canonical form, and (b) canonical form under the lexicographic path ordering

Source: Zhou and Burleson, DAC 1995

Verification of Arithmetic Circuits using Term Rewriting

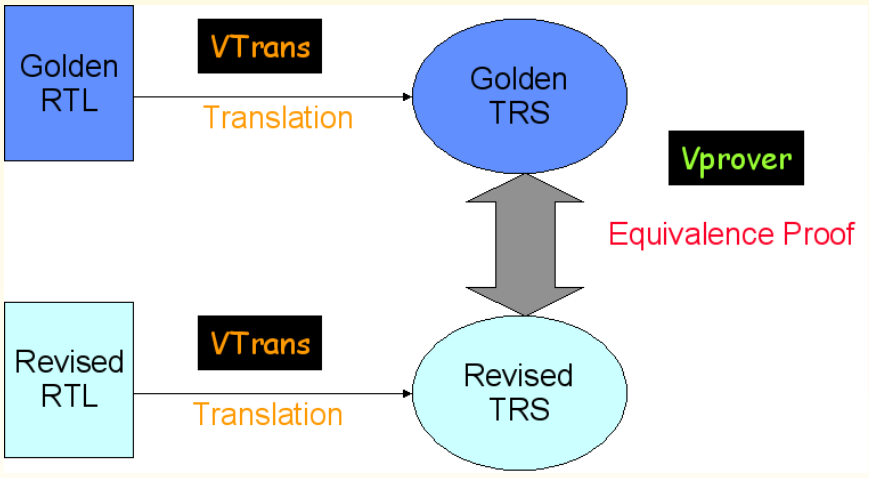
- RTL to RTL equivalence checking
- Verifies large multiplier designs
- Formalism: Term Rewriting Systems

Verifire

- Dedicated Arithmetic Circuit Checker
- Vtrans: Translates Verilog designs to Term Rewriting Systems
- Vprover: Proves equivalence of Term Rewriting Systems
  - Iterative engine which returns error trace if proof not found
  - Maintains an expanding rule base for expression minimization
  - Incomplete, but efficient, engine

S. Vasudevan *et al.* "Automatic Verification of Arithmetic Circuits in RTL using Stepwise Refinement of Term Rewriting Systems," *IEEE Transactions on Computers*, vol. 56, issue 10, pp. 1401-1414, October 2007.

RTL Equivalence Using Term Rewriting Systems (TRS)



Modeling Verilog as TRSs

- Verilog modules translated into *structural* TRS
- Resulting TRS "simulates" Verilog evaluation semantics
- TRS contains symbolic terms for signals in terms of other signals and primary inputs
- Symbolic terms (signal expressions) consist only of RTL operators

Verilog designs

- Every Verilog design corresponds to a TRS
- Every module is a term
- Inputs, Outputs, Reg, Wire, Module instantiations: Subterms
- Variable updating syntactic transformations: Rewrite rules (assignments, case, if-then-else statements)

Equivalence of TRSs

- Observation function applied to both TRSs to obtain *observed* set of terms
- Comparing entire symbolic values of terms: intractable problem
- Compare at intermediate stages of rewriting: comparison points
- Terms compared and expression equivalence proved at every comparison point
- Last comparison point: Normal form

Heuristic for comparison points: compute a partition of the bits for a particular output defined by the assignments to different subsets of bits of the same signal in both the reference (golden) and target designs

Checking Equivalence of Terms (`reduce()`)

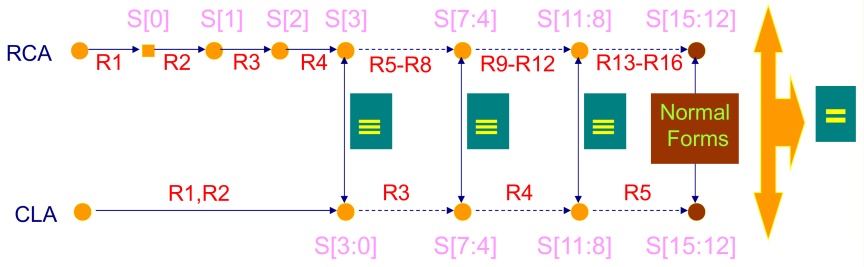
Check for equivalence between two symbolic terms by rewriting based on simplification

$$
\begin{aligned}
(x \;\&\; x) &\rightarrow x \\
((x \;\&\; y)\&z) &\rightarrow (x \;\&\; (y \;\&\; z)) \\
(x << 3) &\rightarrow (x << 2) + (x << 1) + (x << 1) \\
((x << 1) - x) &\rightarrow x \\
((x << 1) << 1) &\rightarrow (x << 2)
\end{aligned}
$$

Equivalence of TRSs Applied to Arithmetic Circuits

- Observed Variables: Outputs
- Comparison points: Points where expressions for partial number of bits is obtained
- Bitwise equivalence of observed terms
- Normal form: Entire bitwidth compared

Example: checking ripple-carry adder against carry lookahead adder



Results on Multipliers

Different sizes of Wallace Tree Multipliers (Verilog RTL) compared with a simple Golden Multiplier (Verilog RTL) of the same size

Compare Verifire against Commercial Tools

Wallace Tree	Verifire	Commercial Tool 1	Commercial Tool 2
4x4	14s	10s	9s
8x8	18s	18s	16s
16x16	25s	unfinished	unfinished
32x32	40s	unfinished	unfinished
64x64	60s	unfinished	unfinished

Distribution of Rewrite Rules for Multipliers Used by `reduce()`

Rule class	Number of rules	Example
Boolean	32	`(x | (y & z)) → ((x | y) & (x | z))`
Add/Subtract	44	`(x + (y - z)) → ((x + y) - z)`
Shift	16	`((x << 1) << 1) → (x << 2)`
Multiplier Specific	9	`((x << 1) - x) → x`
Total	101	

Comparison of Verifire Against Commercial Checker

Multiplier	Verifire (Booth)	Commercial Tool (Booth)	Verifire (Wallace)	Commercial Tool (Wallace)	Verifire (Dadda)	Commercial Tool (Dadda)
$4b \times 4b$	16s	12s	14s	10s	13s	8s
$8b \times 8b$	19s	20s	18s	20s	17s	17s
$16b \times 16b$	24s	1942s	25s	972s	29s	not completed
$32b \times 32b$	37s	not completed	40s	not completed	51s	not completed
$64b \times 64b$	53s	-	60s	-	83s	-

The commercial equivalence checker was assisted by manual compare points (determined from the automatically extracted compare points in Verifire)

Use of TRS with SMT for Verifying Embedded Software

- Verify that two short code segments compute the same result
- Symbolically simulate modern VLIW
- Use TRS to simplify symbolic expressions
- Send query to decision procedure for proof
  - Verify equivalence between two code segments
  - Check conditions of rules to simplify memory term/expression

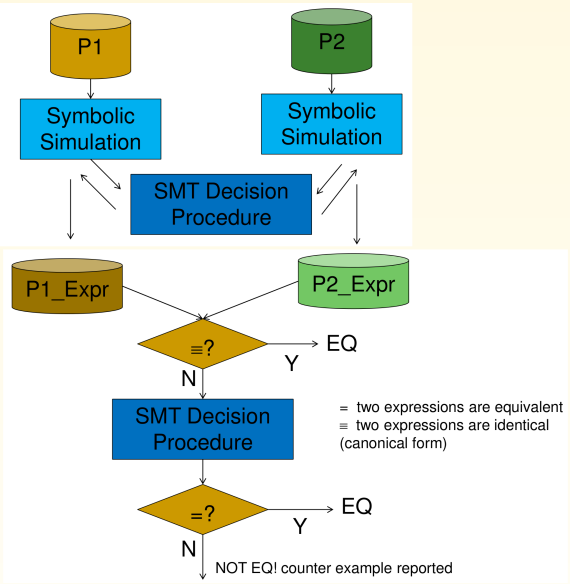
Dealing with Absence of Canonical Forms

- Difficult to reduce two equivalent symbolic expressions to a canonical form
  - If two program segments have different control flows, the expressions will be very different
- Solution: Use a decision procedure with SMT solvers

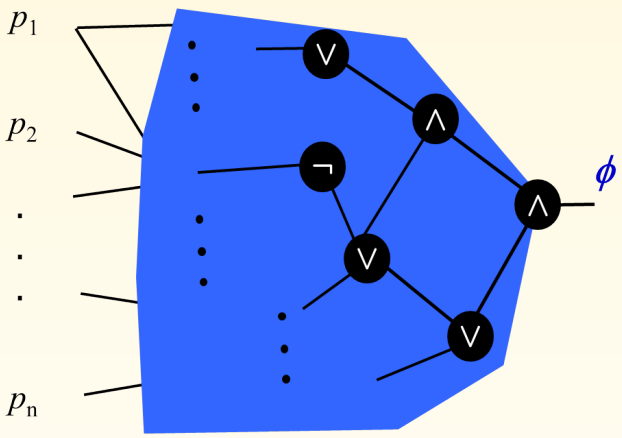
Applied to TI C62x VLIW DSP (can handle DSP assembly code)
Found mismatch in a packet example in TI CPU and ISA reference

References: Feng and Hu, EMSOFT 2005, LCTES 2002; Currie, Feng, Fujita, Hu, Kwan and Rajan, IJPP 2006; Currie, Hu, Rajan and Fujita, DAC 2000

Flow of the Technique



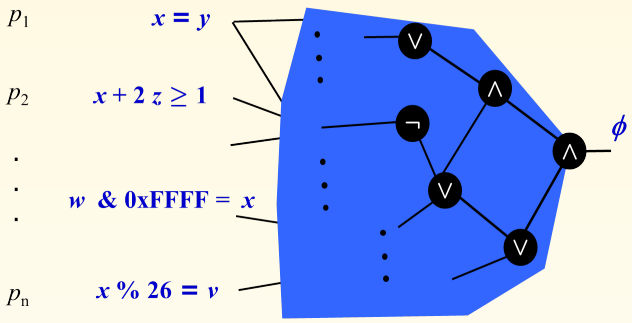
Boolean Satisfiability (SAT)



Is there an assignment to the $p_1, p_2, \ldots, p_n$ variables such that ϕ evaluates to 1?

Source: Barrett and Seshia, ICCAD tutorial, 1999

Satisfiability Modulo Theories



Is there an assignment to the x, y, z, w variables such that ϕ evaluates to 1?

Source: Barrett and Seshia, ICCAD tutorial, 1999

SMT Tools and Constraints

Many Tools

- MathSAT
- MiniSmt
- Boolector
- SMT-RAT
- Yices

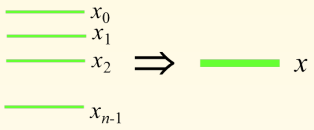
Quantifier-Free Subset of Logic

- Generally deal with formulas of first-order logic without quantifiers ($\forall, \exists$)

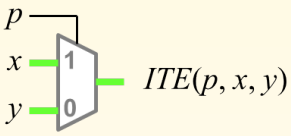
Theory of Equality and Uninterpreted Functions (EUF)

- Only property required is *Congruence*
- $x = y \implies f(x) = f(y)$

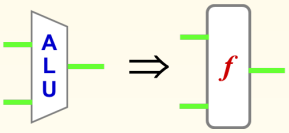
Data and Function Abstraction with EUF



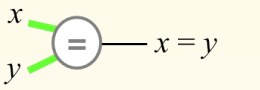
Bit-vectors to Abstract Domain (e.g. $\mathbb{Z}$)



If-then-else



Functional units to Uninterpreted Functions



Test for equality

$$a = x \ \wedge \ b = y \ \Rightarrow \ f(a,b) = f(x,y)$$

Source: Barrett and Seshia, ICCAD tutorial, 1999

ECE Department, University of Texas at Austin | Rewriting | Jacob Abraham, February 6, 2020 48 / 1