14. Term Rewriting Systems

Jacob Abraham

Department of Electrical and Computer Engineering
The University of Texas at Austin
Verification of Digital Systems
Spring 2017
March 2, 2017

Outline

- Term Rewriting Systems (TRS)
  - 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 $R_1$: $\text{Gcd}(a, b)$ if $b \neq 0 \Rightarrow \text{Gcd}(b, \text{Rem}(a, b))$
- Rule $R_2$: $\text{Gcd}(a, 0) \Rightarrow a$
- Rule $R_3$: $\text{Rem}(a, b)$ if $a < b \Rightarrow a$
- Rule $R_4$: $\text{Rem}(a, b)$ if $a \geq b \Rightarrow \text{Rem}(a-b, b)$

Example:

$\text{Gcd}(2, 4) \xrightarrow{R_1} \text{Gcd}(4, \text{Rem}(2, 4)) \xrightarrow{R_3} \text{Gcd}(4, 2) \xrightarrow{R_4} \text{Gcd}(2, \text{Rem}(0, 2)) \xrightarrow{R_3} \text{Gcd}(2, 0) \xrightarrow{R_1} 2$

Simple Arithmetic Rewriting

Terms

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

Rules

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

Source: Shaun Feng
Example(s) in Rewriting

Example: \((\ast 4 (+ 3 3) \equiv (\ast (+ 4 4) 3)\)

\[\begin{align*}
(\ast 4 (+ 3 3) &\rightarrow (\ast(\ast 3 2)) \rightarrow (\ast 4 (\ast 2 3)) \rightarrow (\ast(\ast 2 3) 4) \\
(\ast (+ 4 4) 3 &\rightarrow (\ast (\ast 4 2) 3) \rightarrow (\ast (\ast 2 4) 3) \rightarrow (\ast(\ast 2 3) 4)
\end{align*}\]

Prove if \((\ast (x + y) y) \equiv (\ast (x x) y)\)

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 \((R_1, R_2, \ldots)\)
- **R**: Set of labeled rules (may be conditional) \((r_1, r_2, \ldots, r_n)\)

Rewrite process

\[t_1 \xrightarrow{r_1} t_2 \xrightarrow{r_2} t_3 \xrightarrow{r_k} \ldots \xrightarrow{r_m} t_n \text{ (Normal Form)}\]

- 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, ∧, ¬

Rules
- Rule R1: \( a \land b \rightarrow \neg 2\text{NAND}(a, b) \)
- Rule R2: \( \neg(\neg a) \rightarrow a \)
- Rule R3: \( \neg a \rightarrow 2\text{NAND}(a, a) \)

Apply the rules to get a 3NAND

\[ \neg((A \land B) \land C) \xrightarrow{R1} \neg(\neg 2\text{NAND}(A \land B, C)) \xrightarrow{R2} 2\text{NAND}(A \land B, C) \xrightarrow{R3} 2\text{NAND}(2\text{NAND}(A, B), 2\text{NAND}(A, B)), C \]

Termination and Cofluence

Termination
- No infinite rewriting sequence \( \rightarrow \) normal form exists

Cofluence
- Terms can be rewritten in multiple ways, but will eventually yield the same results
  - \( *(+ 2 1) (+ 3 4) \rightarrow *(3 (+ 3 4)) \rightarrow *(3 7) \)
  - \( *(+ 2 1) (+ 3 4) \rightarrow *(+ 2 1) 7 \rightarrow *(3 7) \)
- 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:** \( t \rightarrow t \)
- **(R) Replacement:**
  - R1: \( a \rightarrow a - 3 \) if \( a \in I \) and \( a \geq 3 \)
  
  \[
  t_k(x) \rightarrow t_n(x) \\
  t_k(x_0/x) \rightarrow t_n(x_0/x)
  \]
- **(C) Congruence**
  - R1: \( a \rightarrow a - 3 \) if \( a \geq 3 \)
  
  \[
  t_1 \rightarrow t'_1, \ldots, t_k \rightarrow t'_m \\
  f(t_1, \ldots, t_k) \rightarrow f(t'_1, \ldots, t'_k)
  \]
- **(T) Transitivity**
  
  \[
  t_1 \rightarrow t_2, \ldots, t_2 \rightarrow t_3 \\
  t_1 \rightarrow t_3
  \]

### TRS Closure

- **One step rewrite:**
  - Reflexivity
  - Congruence
  - Replacement

Source: Shaun Feng

Equivalence Under TRS

- For each pair of compare points (P1, P2), from the models M1 and M2
  - Get normal forms by applying rewrite rules
  - If normal_form(P1) and normal_form(P2) are the same term, then equivalence is proven

Assumptions necessary for TRS
- Convergence
- Canonical forms exist for equivalent terms

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


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)

**Verification Algorithm**

```plaintext
main (vG, vR) {
    trsG := translate (vG)
    trsR := translate (vR)
    proofOutcome := prove (trsG, trsR)
}

prove (trsG, trsR) {
    CP := computeComparePoints (trsG, trsR)
    for (every comparison point (cG, cR) ∈ CP)
        if (reduce (cG) is not equal to reduce (cR))
            return failure
    return success
}

reduce (t) {
    while (some rule can be applied)
        rewrite (t)
}
```

Department of Electrical and Computer Engineering, The University of Texas at Austin
J. A. Abraham, March 2, 2017
Verilog to TRS Translation (translate())

- Translate source Verilog modules into a structural TRS that “simulates” the Verilog evaluation semantics
- Structural TRS is at the same level of abstraction as the Verilog design
- Result is a rewrite system which can be used to compute the symbolic term for any signal in terms of other signals and/or primary inputs
- Each hierarchical signal is represented by a new constant function symbol (signal function)(hierarchy is “flattened”)
- Rewrite rules rewrite each signal function into an expression consisting of RTL operators and other signal functions

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 ($\text{reduce()}$)

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

\[
\begin{align*}
(x & x) & \rightarrow x \\
((x & y) & z) & \rightarrow (x & (y & z)) \\
(x << 3) & \rightarrow (x << 2) + (x << 1) + (x << 1) \\
((x << 1) - x) & \rightarrow x \\
((x << 1) << 1) & \rightarrow (x << 2)
\end{align*}
\]

Verilog for Ripple-Carry Adder

```verilog
module rcal6bit(A, B, Cin, S, Cout);
  input [15:0] A, B;
  input Cin;
  output [15:0] S;
  output Cout;
  reg S, Cout;
  wire [14:0] Carry;

  rcalbit rcalbit0(A[0], B[0], Cin, S[0], Carry[0]); R1
  rcalbit rcalbit1(A[1], B[1], Carry[0], S[1], Carry[1]); R2
  ;
  rcalbit rcalbit15(A[15], B[15], Carry[14], S[15], Cout); R16
endmodule

module rcalbit(A, B, C, S, Cout);
  input A, B, C;
  output S, Cout;
  assign S = A & B & C;
  assign Cout = A&B | B&C | C&A;
endmodule
```

Department of Electrical and Computer Engineering, The University of Texas at Austin
J. A. Abraham, March 2, 2017
Example of Rules in the Structural TRS for the Adder

R1: rcalbit0.A() → A[0]()
rcalbit0.B() → B[0]()
rcalbit0.C() → Cin()
rcalbit0.S() → (rcalbit0.A() ∧ rcalbit0.B() ∧ rcalbit0.C())
rcalbit0.Cout() → ((rcalbit0.A() & rcalbit0.B()) | (rcalbit0.B() & rcalbit0.C()) | (rcalbit0.C() & rcalbit0.A()))
S[0]() → rcalbit0.S()
Carry[0]() → rcalbit0.Cout()

R2: rcalbit1.A() → A[1]()
rcalbit1.B() → B[1]()
rcalbit1.C() → Carry[0]()

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

<table>
<thead>
<tr>
<th>Wallace Tree</th>
<th>Verifire</th>
<th>Commercial Tool 1</th>
<th>Commercial Tool 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>4x4</td>
<td>14s</td>
<td>10s</td>
<td>9s</td>
</tr>
<tr>
<td>8x8</td>
<td>18s</td>
<td>18s</td>
<td>16s</td>
</tr>
<tr>
<td>16x16</td>
<td>25s</td>
<td>unfinished</td>
<td>unfinished</td>
</tr>
<tr>
<td>32x32</td>
<td>40s</td>
<td>unfinished</td>
<td>unfinished</td>
</tr>
<tr>
<td>64x64</td>
<td>60s</td>
<td>unfinished</td>
<td>unfinished</td>
</tr>
</tbody>
</table>

Distribution of Rewrite Rules for Multipliers Used by reduce()

<table>
<thead>
<tr>
<th>Rule class</th>
<th>Number of rules</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>32</td>
<td>((x \land (y \land z)) \rightarrow ((x \land y) \land (x \land z)))</td>
</tr>
<tr>
<td>Add/Subtract</td>
<td>44</td>
<td>((x + (y - z)) \rightarrow ((x + y) - z))</td>
</tr>
<tr>
<td>Shift</td>
<td>16</td>
<td>((x &lt;&lt; 1) &lt;&lt; 1 \rightarrow (x &lt;&lt; 2))</td>
</tr>
<tr>
<td>Multiplier Specific</td>
<td>9</td>
<td>((x &lt;&lt; 1) - x \rightarrow x)</td>
</tr>
<tr>
<td>Total</td>
<td>101</td>
<td></td>
</tr>
</tbody>
</table>
Comparison of Verifire Against Commercial Checker

<table>
<thead>
<tr>
<th>Multiplier</th>
<th>Verifire (Booth)</th>
<th>Commercial Tool (Booth)</th>
<th>Verifire (Wallace)</th>
<th>Commercial Tool (Wallace)</th>
<th>Verifire (Dadda)</th>
<th>Commercial Tool (Dadda)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4b x 4b</td>
<td>16s</td>
<td>12s</td>
<td>14s</td>
<td>10s</td>
<td>13s</td>
<td>8s</td>
</tr>
<tr>
<td>8b x 8b</td>
<td>19s</td>
<td>20s</td>
<td>18s</td>
<td>20s</td>
<td>17s</td>
<td>17s</td>
</tr>
<tr>
<td>16b x 16b</td>
<td>24s</td>
<td>1942s</td>
<td>25s</td>
<td>972s</td>
<td>29s</td>
<td>29s</td>
</tr>
<tr>
<td>32b x 32b</td>
<td>37s</td>
<td>not completed</td>
<td>40s</td>
<td>not completed</td>
<td>51s</td>
<td>83s</td>
</tr>
<tr>
<td>64b x 64b</td>
<td>53s</td>
<td>-</td>
<td>60s</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

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

Flow of the Technique

Boolean Satisfiability (SAT)

Is there an assignment to the \( p_1, p_2, \ldots, p_n \) variables such that \( \phi \) evaluates to 1?

Source: Barrett and Seshia, ICCAD tutorial, 1999
Is there an assignment to the $x, y, z, w$ variables such that $\phi$ evaluates to 1?

Source: Barrett and Seshia, ICCAD tutorial, 1999
**Data and Function Abstraction with EUF**

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

\[
x_0, x_1, x_2, \ldots, x_{n-1} \Rightarrow x
\]

Functional units to Uninterpreted Functions

\[
a = x \land b = y \Rightarrow f(a,b) = f(x,y)
\]

Source: Barrett and Seshia, ICCAD tutorial, 1999

**Hardware Abstraction with EUF**

Source: Barrett and Seshia, ICCAD tutorial, 1999
### Software Based Self Test

**Advantages**
- Minimized DFT circuitry
- Reduced external tester performance
- Excessive test power and over-testing eliminated

### RT Level Test Generation for Hard-to Detect Faults

**Overview**
- Map gate level stuck-at fault to RTL
- Capture the propagation constraints as an LTL property
- Generate a witness for the LTL property using Bounded Model Checking
- All required constraints available in RTL
- Use SMT based Bounded Model Checking
- Scaling with cone-of-influence reduction

---

Prabhu et al., ETS 2012
RTL Test Generation for Hard-to-Detect Faults

**Experimental Setup**
- OR1200 RISC processor was DUT
- EBMC Model checker / Boolector SMT solver
- Bound of pipeline depth + 1
- Focused on hard to detect faults in control logic
- Commercial ATPG to sieve out easy to detect stuck-at faults
- 78% Fault coverage by commercial ATPG

**Coverage and Run Time Comparisons** *(Prabhu et al., 2012)*
Using TRS to Design and Verify Processors

Use TRS as a Hardware Description Language

- Can be used for clean, expressive, precise and concise descriptions of microarchitectures, memory models and cache coherence protocols
- Correctness of a TRS can be verified against a reference TRS specification
- HDL (e.g., Verilog) description can be synthesized from the TRS description

Commercialization

- Bluespec, Inc., Synthesis tool
- Tool uses guarded, atomic actions


Example: Hardware Synthesis of GCD

GCD(x, y) if x < y → GCD(y, x)
GCD(x, y) if x ≥ y and y ≠ 0 → GCD(x-y, y)

Verilog synthesized from TRS: 40.1 MHz and used 24% of FPGA
Hand written Verilog: 53 MHz and used 16% of the same FPGA
(IEEE Micro, May-June 1999)