Vdiff: A Program Differencing Algorithm for Verilog HDL

Adam Duley  
ARM Inc

Christopher Spandikow  
IBM Corporation

Miryung Kim  
The University of Texas at Austin
Problem: Limitations of using *diff* on evolving hardware designs

- assumes sequential execution semantics
- relies on code elements having unique names
- does not leverage Boolean expression equivalence checking despite the availability of SAT solvers
Solution: Vdiff

- a position-independent differencing algorithm with intimate knowledge of Verilog semantics
- 96.8% precision with 97.3% recall compared to manually classified differences
- produces syntactic differencing results in terms of Verilog-specific change types
Outline

• **Motivation**
• Verilog Background
• Vdiff Algorithm
• Evaluation
• Conclusions
Motivation

- hardware designers collaboratively evolve large Verilog programs
- hard to use diff-like tools during code reviews
- develop a foundation for reasoning about evolving hardware design descriptions
include "uart_defines.v"
module uart_rfifo (clk, reset, data_out);
input clk, reset;
output [fifo_width-1:0] data_out;
reg [fifo_counter_w-1:0] fifo;
wire [fifo_pointer_w-1:0] overrun;
always @(posedge clk or posedge reset)
begin
  if(reset)
    begin
      fifo[1] <=  0;
      fifo[0] <=  0;
    end
end
assign data_out = fifo[0];
endmodule
include "uart_defines.v"

module uart_rfifo (clk, reset, data_out);
input clk, reset;
output [fifo_width-1:0] data_out;
reg [fifo_counter_w-1:0] fifo;

always @(posedge clk or posedge reset)
begin
  if(reset)
  begin
    fifo[1] <= 0;
    fifo[0] <= 0;
  end
end
assign data_out = fifo[0];
endmodule

Modules are building blocks with an explicit input and output port interface.

A module is similar to a Java class.
```verilog
include "uart_defines.v"

module uart_rfifo (clk, reset, data_out);
input clk, reset;
output [fifo_width-1:0] data_out;
reg [fifo_counter_w-1:0] fifo;
wire [fifo_pointer_w-1:0] overrun;

always @(posedge clk or posedge reset)
begin
if(reset)
begin
    fifo[1] <= 0;
    fifo[0] <= 0;
end
assign data_out = fifo[0];
endmodule

Input and output ports are public interfaces that connect modules to external hierarchy.

They are similar to a constructor’s parameter list in Java.
```
include "uart_defines.v"
module uart_rfifo (clk, reset, data_out);
input clk, reset;
output [fifo_width-1:0] data_out;
reg [fifo_counter_w-1:0] fifo;
wire [fifo_pointer_w-1:0] overrun;
always @(posedge clk or posedge reset)
begin
  if(reset)
  begin
    fifo[1] <=  0;
    fifo[0] <=  0;
  end
assign data_out = fifo[0];
endmodule

Wires, registers, & integers are variable declarations.
module uart_rfifo (clk, reset, data_out);
input clk, reset;
output [fifo_width-1:0] data_out;
reg [fifo_counter_w-1:0] fifo;
wire [fifo_pointer_w-1:0] overrun;
always @(posedge clk or posedge reset)
begin
    if(reset)
    begin
        fifo[1] <= 0;
        fifo[0] <= 0;
    end
end // always
assign data_out = fifo[0];
endmodule
Verilog HDL Background

```verilog
include "uart_defines.v"
module uart_rfifo (clk, reset, data_out);
  input clk, reset;
  output [fifo_width-1:0] data_out;
  reg [fifo_counter_w-1:0] fifo;
  wire [fifo_pointer_w-1:0] overrun;
  always @(posedge clk or posedge reset)
    begin
      if(reset)
        begin
          fifo[1] <= 0;
        end
    end // always
  assign data_out = fifo[0];
endmodule
```

Assign statements model concurrent dataflow.
include "uart_defines.v"
module uart_rfifo (clk, reset, data_out);
input clk, reset;
output [fifo_width-1:0] data_out;
reg [fifo_counter_w-1:0] fifo;
wire [fifo_pointer_w-1:0] overrun;
always @(posedge clk or posedge reset)
begin
  if(reset)
  begin
    fifo[1] = 0;
    fifo[0] = 0;
  end
end
assign data_out = fifo;
endmodule

Blocking statements are sequential assignments.
include "uart_defines.v"

module uart_rfifo (clk, reset, data_out);
input clk, reset;

output [fifo_width-1:0] data_out;

reg [fifo_counter_w-1:0] fifo;

wire [fifo_pointer_w-1:0] overrun;

always @(posedge clk or posedge reset)
begin
if(reset)
begin
  fifo[1] <= 0;
  fifo[0] <= 0;
end
end

assign data_out = fifo;
endmodule

Non-blocking statements are concurrent assignments, and they are prevalent in Verilog designs.
Verilog’s non-unique identifiers and concurrent semantics cause `diff` to identify a large amount of false positives.
Outline

• Motivation
• Verilog Background
• Vdiff Algorithm
• Evaluation
• Conclusions
Algorithm

• input: two versions of a Verilog file
• output: syntactic differences in terms of change types
  1. extract Abstract Syntax Tree (AST) from each file
  2. compare the two trees
  3. filter false positives in changes to sensitivity lists using a SAT solver
  4. categorize differences based on Verilog syntax
module uart_rfifo (clk, reset, data_out, overrun);
always @(posedge clk)
begin
  if(reset)
  begin
    fifo[1] <= 0;
    fifo[0] <= 0;
  end
end // always
always @(posedge clk)
begin
  if (reset)
    overrun <= 0;
end // always
assign data_out = fifo[0];
endmodule
Tree Differencing Algorithm

- hierarchically compare tree nodes from the top down
- initially align nodes using the longest common subsequence (LCS) algorithm—unmatched nodes are split into ADD and DELETE sets
- for each pair in ADD x DELETE, calculate the textual similarity
- use greedy weighted bipartite graph matching to associate a DELETE node to a corresponding ADD node
module

always (fifo)

assign

if

<= fifo[1]

<= fifo[0]

<= overrun

always (run)

module

always (fifo)

assign

if

<= fifo[0]

<= fifo[1]

<= fifo[2]

<= overrun

<= overrun

uart_rfifo.v rev 87

uart_rfifo.v rev 88
```verilog
module always (fifo)
always (run)
assign <= overrun
if <= fifo[1] <= fifo[0]
```

```verilog
module always (run)
always (fifo)
assign <= overrun
if <= fifo[0] <= fifo[2]
if <= fifo[1]
```
module

always (fifo)

always (run)

assign

if

<= fifo[1]

<= fifo[0]

<= overrun

uart_rfifo.v rev 87

uart_rfifo.v rev 88
Tree Comparison

uart_rfifo.v rev 87

module
always (fifo)
always (run)
assign
if
<= fifo[1]
<= fifo[0]
<= overrun

uart_rfifo.v rev 88

module
always (fifo)
always (run)
assign
if
<= fifo[0]
<= fifo[1]
<= fifo[2]
<= overrun

?
Tree Comparison

uart_rfifo.v rev 87

module
always (fifo)
assign
if
<= fifo[0]
<= fifo[1]
if
overrun

uart_rfifo.v rev 88

module
always (fifo)
assign
if
<= fifo[1]
<= fifo[2]
if
overrun
<= fifo[0]
<= fifo[1]
Tree Comparison

uart_rfifo.v rev 87

module
always (fifo)
  assign
   if
    <= fifo[1]
  assign
   if
    <= fifo[0]
  assign
   if
    <= overrun
always (run)
assign
if
<= fifo[1]
if
<= fifo[0]
if
<= overrun

uart_rfifo.v rev 88

module
always (fifo)
  assign
   if
    <= fifo[1]
  assign
   if
    <= fifo[0]
  assign
   if
    <= fifo[2]
always (run)
assign
if
<= fifo[0]
if
<= fifo[1]
if
<= overrun

LCS Match
mapped
add
delete
module
always (fifo)
assign
if
<= fifo[1]
<= fifo[0]
<= overrun
always (run)
assign
if
<= fifo[0]
<= fifo[1]
<= fifo[2]
module
always (fifo)
assign
if
<= fifo[0]
<= fifo[2]
<= fifo[1]
always (run)
assign
if
<= fifo[1]
<= overrun
<= fifo[0]
module
always (fifo)
assign
if
<= fifo[0]
<= fifo[1]
always (run)
if
<= overrun
uart_rfifo.v rev 87
uart_rfifo.v rev 88
module
always (run)
assign
if
<= overrun
<= fifo[0]
<= fifo[1]
<= fifo[2]
Tree Comparison

uart_rfifo.v rev 87

module
always (fifo)
assign
if
<= fifo[1]
<= fifo[0]
<= overrun

uart_rfifo.v rev 88

module
always (fifo)
assign
if
<= fifo[0]
<= fifo[1]
<= fifo[2]

mapped
LCS Match
add
delete
uart_rfifo.v rev 87-88 diff

module
always (fifo)
if
<= fifo[2]

Line 221, NB_ADD, A Non-Blocking assignment has been added
Boolean Equivalence Check

always @(posedge clk or negedge reset)
  A = clk | ! reset

always @(negedge reset or posedge clk)
  B = ! reset or clk

SAT Solver

\[(A \& ! B) \mid (B \& !A) \neq 1\]

Solvable?

- Yes, then A is not equivalent to B.
- No, then A is equivalent to B.
## Change Type Classification

<table>
<thead>
<tr>
<th>Syntactic Elements</th>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Blocking</td>
<td>NB_ADD</td>
<td>Non-Blocking assignment added</td>
</tr>
<tr>
<td></td>
<td>NB_RMV</td>
<td>Non-Blocking assignment removed</td>
</tr>
<tr>
<td></td>
<td>NB_CE</td>
<td>Non-Blocking assignment changed</td>
</tr>
<tr>
<td>Always</td>
<td>AL_ADD</td>
<td>Always block added</td>
</tr>
<tr>
<td></td>
<td>AL_RMV</td>
<td>Always block removed</td>
</tr>
<tr>
<td></td>
<td>AL_SE</td>
<td>Changes in the sensitivity list</td>
</tr>
</tbody>
</table>
Vdiff Tool Implementation

- Eclipse plugin based on open source *veditor* plug-in
- interfaces with Eclipse *compare* plug-in
- integrates with SVN (*subclipse* plug-in)

[http://users.ece.utexas.edu/~miryung/software/Vdiff/web/index.html](http://users.ece.utexas.edu/~miryung/software/Vdiff/web/index.html)
Vdiff Eclipse Plugin

Syntactic Diff Window

Textual Diff Window

Change Type Classification
Outline

• Motivation
• Verilog Background
• Vdiff Algorithm
• **Evaluation**
• Conclusions
Evaluation

• compare Vdiff’s results with manual differencing results— the first two authors manually inspected the revisions

• subjects
  – UART16550 (opencores.org)
  – RAMP GateLib DRAM controller (ramp.eecs.berkeley.edu)

• Criteria: precision & recall
We evaluated 210 Verilog file revisions with more than 1000 differences across two different projects.
Findings

• Vdiff matches position-independent constructs very well
• Vdiff struggles on line edits with low text similarity
• Feedback from a logic designer at IBM:
  “I can see a use for [the change-types] right away. It would be great for team leads because they could look at this log of changes and understand what has changed between versions without having to look at the files [textual differences].”
Example: Expected Results

/* Old */

/* If Condition Changed */
if (!srx_pad_i)
    counter_b <= 0x191;

/* New */

/* If Condition Changed */
if (!srx_pad_i || rstate == sr_idle)
    counter_b <= 0x191;
Example: Vdiff Results

/* Old */

/* IF Block Removed*/
if (!srx_pad_i)
    counter_b <= 0x191;

/* New */

/* IF Block Added */
if (!srx_pad_i || rstate == sr_idle)
    counter_b <= 0x191;
Comparison of AST Matching Algorithms

- Exact matching [Neamtiu 2005]
- In-order matching [Cottrell 2007]
- Greedy weighted bipartite matching [Vdiff]
Comparison of AST Matching Algorithms

- **Exact matching [Neamtiu 2005]**
- In-order matching [Cottrell 2007]
- Greedy weighted bipartite matching [Vdiff]
Comparison of AST Matching Algorithms

- Exact matching [Neamtiu 2005]
- In-order matching [Cottrell 2007]
- Greedy weighted bipartite matching [Vdiff]
Comparison of AST Matching Algorithms

- Exact matching [Neamtiu 2005]
- In-order matching [Cottrell 2007]
- **Greedy weighted bipartite matching [Vdiff]**
Comparison of AST Matching Algorithms

- Exact matching [Neamtiu 2005]
- In-order matching [Cottrell 2007]
- Greedy weighted bipartite matching [Vdiff]

<table>
<thead>
<tr>
<th></th>
<th>Average Match</th>
<th>Exact Match</th>
<th>In-Order Match</th>
<th>Weighted Bipartite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precision</td>
<td></td>
<td>56.1%</td>
<td>90.9%</td>
<td>97.5%</td>
</tr>
<tr>
<td>Recall</td>
<td></td>
<td>67.9%</td>
<td>91.8%</td>
<td>97.7%</td>
</tr>
</tbody>
</table>
Comparison of AST Matching Algorithms

1097 differences from 210 file revisions in 2 real world projects shows that *the ordering of code actually matters in practice* when it comes to computing differences.

<table>
<thead>
<tr>
<th></th>
<th>Average Match Precision</th>
<th>Exact Match Precision</th>
<th>In-Order Match Precision</th>
<th>Weighted Bipartite Match Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precision</td>
<td>56.1%</td>
<td>90.9%</td>
<td>97.5%</td>
<td></td>
</tr>
<tr>
<td>Recall</td>
<td>67.9%</td>
<td>91.8%</td>
<td>97.7%</td>
<td></td>
</tr>
</tbody>
</table>
Comparison with General Model Differencing Framework

• EMF [Eclipse EMF compare project]
  – Mapped (1) modules to classes, (2) always blocks and continuous assignments to operations, (3) wires, registers, and ports to fields, and (4) modular instantiations to reference pointers in an EMF ecore model.
  – Results (Recall=47%, Precision=80%) show a need to expand the Ecore model to be able to handle specific concurrency constructs and non-unique identifiers.

• Sidiff [Treude et al., 2007, Schmidt and Gloetzner, 2008]
  – At the time of our evaluation Sidiff did not provide APIs to allow us to map Verilog language constructs to their general differencing algorithms.
Discussion

• Vdiff is sensitive to subtle changes to variable names and IF-conditions
  – Further investigation of different name similarity measures is required
• renaming of wires, registers, and modules caused false positives
• Vdiff’s algorithm currently cannot recover from node mismatches
• equivalence check using a SAT solver is limited to sensitivity lists
Outline

• Motivation
• Verilog Background
• Vdiff Algorithm
• Evaluation
• Conclusions
Related Work

• **Syntactic program differencing**
  – Vdiff is similar to these but identifies syntactic differences robustly even when multiple AST nodes have similar labels and when they are reordered.

• **Model differencing**
  – UMLdiff [Xing and Stroulia 2005], Sidiff [Kelter et al.] and EMF [Eclipse EMF]

• **Change types**
  – Change Distiller [Fluri et al. 2007]
  – Verilog change types [Sudakrishnan et al. 2009]

• **Differential symbolic execution** [Person et al. 2008]
Conclusions

• *Vdiff* is a position-independent differencing algorithm designed for hardware design descriptions
  – computes syntactic differences with high recall (96.8%) and high precision (97.3%)
  – classifies differences in terms of Verilog specific change types
  – can enable analysis of evolving hardware design
Acknowledgment

Vdiff website:
http://www.ece.utexas.edu/~miryung/software/Vdiff/web/index.html

The authors thank Greg Gibeling and Dr. Derek Chiou for providing accesses to the RAMP repository and Dr. Adnan Aziz and anonymous reviewers for their detailed comments on our draft.
Questions
Backup