Noname manuscript No.
(will be inserted by the editor)

Vdiff: A Program Differencing Algorithm for Verilog Hardware Description Language

Adam Duley · Chris Spandikow · Miryung Kim

The second revision is submitted to the ASEJ special issue on selected articles from the ASE conference on March 15th, 2012.

Abstract During code review tasks, comparing two versions of a hardware design description using existing program differencing tools such as *diff* is inherently limited because these tools implicitly assume sequential execution semantics, while hardware description languages are designed to model concurrent computation. We designed a position-independent differencing algorithm to robustly handle language constructs whose relative orderings do not matter. This paper presents *Vdiff*, an instantiation of this position-independent differencing algorithm for Verilog HDL. To help programmers reason about the differences at a high-level, Vdiff outputs syntactic differences in terms of Verilog-specific change types. The quantitative evaluation of Vdiff on two open source hardware design projects shows that Vdiff is very accurate, with overall 96.8% precision and 97.3% recall when using manually classified differences as a basis of comparison. Vdiff is also fast and scalable—it processes the entire revision history of nine open projects all under 5.35 minutes. We conducted a user study with eight hardware design experts to understand how the program differences identified by the experts match Vdiff's output. The study results

This research is in part supported by National Science Foundation grants under CCF-1043810, CCF-1117902, and CCF-1149391 and 2011 Microsoft SEIF Award. The research was in part conducted while the first two authors were graduate students at The University of Texas at Austin.

Adam Duley
ARM Inc.
E-mail: adam.duley@arm.com

Chris Spandikow
IBM Corporation.
E-mail: spandiko@gmail.com

Miryung Kim
Electrical and Computer Engineering, The University of Texas at Austin E-mail: miryung@ece.utexas.edu

show that Vdiff's output is better aligned with the experts' classification of Verilog changes than an existing textual program differencing tool.

Keywords software evolution · program differencing · hardware description languages

1 Introduction

Hardware description languages (HDLs) are pervasively used by engineers to abstractly define hardware circuitry. Verilog, one of the most widely used HDLs, uses a C-like syntax to describe massively concurrent tasks—Verilog statements can represent parallel execution threads, propagation of signals, and variable dependency [31]. Hardware projects are in a constant state of change during the development process due to new feature requests, bug fixes, and demands to meet power reduction and performance requirements. During *code review* tasks, hardware engineers predominantly rely on *diff*, which computes line-level differences per file based on a textual representation of a program.

Using existing program differencing tools for Verilog programs has several limitations. First, line-based differencing tools for Verilog programs report many false positive differences because the longest common sequence algorithm [18] maps code in order and thus is too sensitive for languages that model concurrent computation, such as asynchronously-executed statements where the relative ordering between the statements do not matter. This is not only the problem with line-based differencing tools; abstract syntax tree-based differencing algorithms such as Cdiff [35] often match nodes in the same level in order, making it unsuitable for programming languages where concurrent execution is common. Second, unlike Java methods or C functions, Verilog processes such as `always` blocks (i.e., event handlers) do not have unique labels. Thus, existing differencing algorithms such as UMLdiff [34] cannot accurately match position-independent language constructs in out-of-order, when they do not have unique labels that produce one-to-one matching based on name similarity. Third, while Verilog programs frequently use Boolean expressions to define circuitry, existing algorithms do not perform equivalence checking between these Boolean expressions, despite the availability of a mature technology to solve a Boolean formula satisfiability problem.

To overcome these limitations, we have developed Vdiff that uses an intimate knowledge of Verilog syntax and semantics. Our differencing algorithm takes two versions of a Verilog design file and first extracts abstract syntax trees (ASTs). Traversing the trees top-down, for each level, it uses the longest common sequence algorithm to align nodes by the same label. For each mapped node pair from this process, it recursively applies the same LCS-based algorithm to find correspondences between their children nodes. For the remaining unmapped nodes, it then uses a weighted bipartite graph matching algorithm to find out-of-order matching between similar subtrees to handle

position-independent language constructs. To complement syntactic differencing, we also use an off-the-shelf SAT solver to compare the semantics of two Boolean expressions in the process interface description (i.e., the sensitivity list of Verilog's `always` block). Furthermore, to help programmers better understand AST matching results, it outputs differences in terms of Verilog-specific change types (see Section 3.2 for a detail description on change-types). Vdiff is instantiated as an Eclipse plug-in and available for download [15].

We applied Vdiff to two open source project histories and compared its accuracy with manually labeled differences. We also compared our algorithm with three existing AST matching algorithms, measured the types of changes common in Verilog, assessed the impact of using similarity thresholds in matching AST nodes, and measured the scalability and performance of Vdiff using the revision history of nine open source projects from the OpenCores repository.

Moreover, to assess the utility of Vdiff, we conducted a user study with eight expert hardware design engineers from a large multinational semiconductor and processor company. In this study, the participants were given two code review tasks. For each task, the participants inspected changes between the old and new version of a Verilog program and classified the program differences explicitly as semantic or non-semantic differences. After completing a manual inspection of code differences, they reviewed two sets of program differencing outputs using Vdiff and Tkdiff, a popular program differencing tool among hardware design engineers in the same company. The study results show that Vdiff's output is better aligned with the experts' classification of Verilog program changes than Tkdiff. The expert designers also reported that Vdiff makes it easier for them to filter out non-semantic differences caused by reordering code elements. Vdiff helps them to grasp a high level structure of design changes by presenting Verilog-specific change types in a hierarchical order. The user study and the performance assessment are new contributions since our original ASE 2010 conference paper. To the best of our knowledge, our study is the first user study where hardware logic design engineers classified program differences in Verilog and articulated the strengths and limitations of existing program differencing tools for Verilog HDL.

In summary, our paper makes the following contributions:

- Vdiff uses a position-independent differencing algorithm to robustly handle language constructs whose relative orderings do not matter, for example, statements with concurrent execution semantics. The capability to match code fragments in out-of-order is important for other programming languages as well.
- Vdiff produces accurate differencing results with 96.8% precision and 97.3% recall when using manually classified differences as a basis of evaluation.
- Vdiff's output matches the experts' classification of Verilog program differences better than a textual program differencing tool, Tkdiff. [6].

- Vdiff outputs syntactic differencing results in terms of Verilog-specific change types to help hardware designers better understand the program differences.
- Vdiff is scalable and fast. It processes the entire version history of 54017 modified lines from nine open source projects with the total size of 83489 lines of code in their latest version all under 5.35 minutes.
- Our study participants reported that Vdiff robustly recognizes re-arranged code blocks and filters out non-semantic differences.

Vdiff has several implications for the software engineering research community. First, the hardware design industry is facing challenges in evolving existing design artifacts, just as the software industry is facing the problems of evolving software. Yet, support for evolving hardware designs is very limited compared to evolving software. Our goal is to develop a foundation for reasoning about differences in hardware design descriptions to enable various hardware evolution research, such as regression analysis of hardware designs, change impact analysis, etc. Second, the algorithm in Vdiff could be applied to any language that provides ordering-independent language constructs.

The rest of this paper is organized as follows. Section 2 presents a motivating example for Verilog-specific program differencing. Section 3 presents our algorithm. Section 4 describes our evaluation methods and results. Section 5 presents our user study, and Section 6 discusses Vdiff's limitations and threats to validity. Section 7 discusses related work. Section 8 concludes with a discussion of future work.

2 Motivating Example

Verilog is a hardware description language, in which statements and structures map directly to hardware circuitry and its behavior. Because gates operate concurrently [17], Verilog models concurrency explicitly by providing language constructs such as `always` blocks, continuous assignments (`assign`), or non-blocking assignments (`<=`).

To illustrate the key features of Verilog, Figure 1 provides a simple example extracted from the uart_rfifo.v file in the OpenCores UART 16550 project. This code is a simple implementation of a FIFO queue in Verilog. The key items to note in this example are the module, the `always` blocks, the continuous assignments (`assign`), and the non-blocking assignments. The module `uart_rfifo()`'s `input` and `output` declarations define which inputs are required for the module and which output signals it produces. Registers and wires (`reg` and `wire`) can be considered to be field declarations in the module.

Functional specifications can be written either as an initialization block (`initial`), a procedure block (`always`), or continuous assignments (`assign`). Always blocks are process definitions that are re-evaluated when specified event conditions become true. For example, `always @(posedge clk or posedge wb_rst_i)` is evaluated when either the `clk` or `wb_rst_i` signal transitions

Old Program Version

```
‘include "uart_defines.v"
module uart_rfifo (clk, wb_rst_i, data_in, push, pop, data_out, overrun);
input clk;
output [fifo_width-1:0] data_out;
reg  [fifo_counter_w-1:0] count;
wire [fifo_pointer_w-1:0] top_plus_1 = top + 1'b1;
...

always @(posedge clk or posedge wb_rst_i)
begin
      top <= #1 0;
      bottom <= #1 0;
      count <= #1 0;
      fifo[1] <= #1 0;
-     fifo[2] <= #1 0;
   ...
end   // always
- always @(posedge clk or posedge wb_rst_i)
- begin
-  if (wb_rst_i) overrun   <= #1 1'b0;
-  else
-    if (fifo_reset | reset_status)
-      overrun   <= #1 1'b0;
-    else
-      if(push & ~pop & (count==fifo_depth))
-        overrun   <= #1 1'b1;
- end   // always
assign data_out = fifo[bottom];
endmodule
```

New Program Version

```
‘include "uart_defines.v"
module uart_rfifo (clk, wb_rst_i, data_in, push, pop, data_out, overrun);
input clk;
output [fifo_width-1:0] data_out;
reg  [fifo_counter_w-1:0] count;
wire [fifo_pointer_w-1:0] top_plus_1 = top + 1'b1;
...

always @(posedge clk or posedge wb_rst_i)
+ begin
+   if (wb_rst_i)
+    overrun   <= #1 1'b0;
+   else
+     if (fifo_reset | reset_status)
+      overrun   <= #1 1'b0;
+     else
+       if(push & ~pop & (count==fifo_depth))
+         overrun   <= #1 1'b1;
+ end   // always
+ always @(posedge wb_rst_i or posedge clk)
begin
      top <= #1 0;
      bottom <= #1 0;
      count <= #1 0;
+     fifo[2] <= #1 0;
      fifo[1] <= #1 0;
    ...
end   // always
assign data_out = fifo[bottom];
endmodule
```

Fig. 1 Line-level *diff* results and expected differences between two versions of a Verilog program

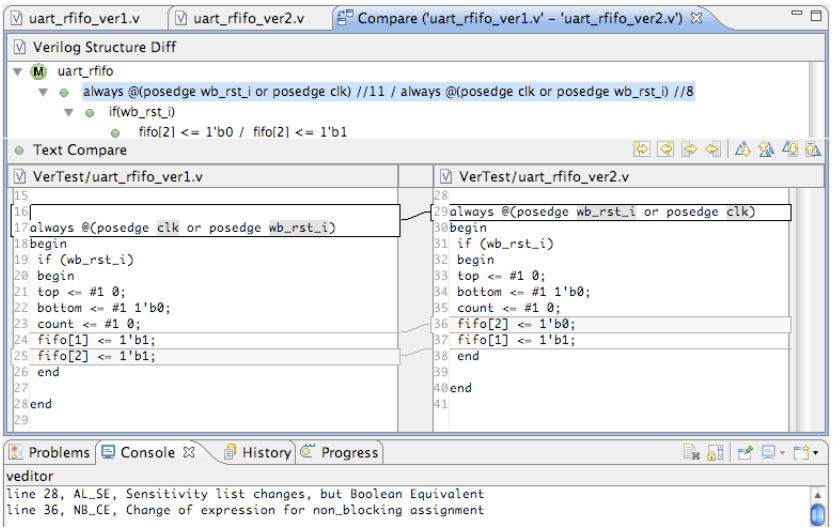

Fig. 2 Vdiff's output in Eclipse IDE

from 0 to 1 due to `posedge`. Verilog provides two different types of assignments, `= and <=`. The blocking assignment (`=`) is similar to an assignment statement in C with sequential execution semantics, while the non-blocking assignment (`<=`) denotes a non-blocking operation that executes simultaneously. In Verilog, non-blocking assignments are generally more common than blocking assignments. Thus, inside the first `always` block, the registers `top`, `bottom`, and `count` are all set to 0 simultaneously, unlike C. In other words, the order in which `top`, `bottom`, and `count` are declared does not matter as long as they are in the same control hierarchy. An ideal program differencing tool for Verilog must not detect such reordering of non-blocking statements, as it does not change the execution semantics. Similarly, the order of the `always` blocks does not matter because all `always` blocks are executed simultaneously. Likewise, all continuous assignments (`assign`) in a module operate concurrently. To draw an analogy between C-like languages and Verilog, one may claim that each `always` block is treated like a function. This idea, however, falls short since multiple blocks can be triggered by the same event list, meaning that one cannot assume that each `always` block has a unique label.

Figure 1 contrasts what a human would consider to be differences and line-level differences computed by *diff*: added lines are marked in red text with `+`, and deleted lines are marked in blue text with `-`. In this code example, the `always` blocks are reordered, the two non-blocking statements are reordered, and the arguments in the first `always` block's sensitivity list are reordered. A human will recognize that, despite textual differences, there are no semantic differences between the two versions. However, *diff* will report several false positives: (1) addition and deletion of the second `always` block, (2) additions and deletions of two non-blocking assignments, and (3) addition and deletion of

the second `always` block's sensitivity list. Furthermore, as *diff* cannot recognize Verilog syntax, it will report differences that do not respect the boundaries of each `always` block.

The following list summarizes the unique characteristics of Verilog from a program differencing perspective.

- Verilog models concurrent executions by using constructs such as non-blocking assignments, processes, and continuous assignments. Thus, ordering-sensitive differencing algorithms designed for sequential execution semantics will report many false positive differences
- In Verilog, processes do not necessarily have unique labels, even though it is thought to be a bad practice to use the same name for multiple processes. Thus, differencing algorithms cannot rely on mapping code elements solely based on name similarity.
- Frequent usage of Boolean expressions in Verilog interface descriptions provides an opportunity to leverage a SAT solver to compare process interfaces (i.e., sensitivity lists).

3 Approach

Vdiff accepts two versions of a Verilog design file and outputs syntactic differences in terms of Verilog-specific change types. It uses a hybrid program differencing approach that performs a syntactic comparison of two abstract syntax trees while checking semantic equivalence in process interface descriptions using an off-the-shelf SAT solver. Section 3.1 discusses our abstract syntax tree matching algorithm that accounts for concurrent execution semantics such as non-blocking assignments. Section 3.2 presents Verilog-specific change types, which are designed to help programmers better understand AST-level matching results. This section also describes when and how our algorithm performs a semantic comparison using a SAT solver. Section 3.3 describes our Vdiff Eclipse plug-in [15].

3.1 Position-Independent Abstract Syntax Tree Differencing

Our algorithm shown in Algorithm 1 takes as input the old and new versions of a Verilog module and two thresholds used for determining text similarity. For each Verilog module, it extracts an abstract syntax tree using the Verilog syntax parser module provided by the VEditor plug-in [7]. Then it marks AST nodes that correspond to non-blocking assignments, continuous assignments, and always blocks. Marking these nodes allows for the matching algorithm to carefully handle semantically equivalent reordering of such nodes. Figure 3 shows an example AST, where unordered children are marked with a dotted edge. The resulting abstract syntax tree allows certain nodes to be arranged in any sequence.

Once the trees, L and R, are built for each file, F_o and F_n, they are compared hierarchically from the top down using `compareTrees()`. The initial comparison is done by aligning nodes in the same level by the same labels, using the longest common subsequence algorithm [18]. Any unmatched node in R is added to ADD, and any unmatched node in L is added to DEL. The step is recursively applied to all children of the matching nodes.

Algorithm 1: Position-Independent AST Matching

```
Input: F_o, F_n /* old and new versions                                   */
th_s, th_l /* similarity thresholds for short text and long text          */
Output: ADD /* a set of nodes added to F_n                               */
DEL /* a set of nodes deleted from F                                      */
MAP /* a set of mapped node pairs                                         */
L := createAST (F_o), R := createAST (F_n)
ADD := ∅, DEL := ∅, MAP := ∅, Candidate := ∅
compareTrees (L, R, MAP, ADD, DEL)
findCandidate (ADD, DEL, Candidate, th_l, th_s)
repeat
    /* Identify a weighted bipartite matching by selecting a candidate
       match with the highest likeness value and updating ADD and DEL
       accordingly                                                        */
    Candidate = sort(Candidate)
    foreach p ∈ Candidate do
        if p.a ∈ ADD and p.d ∈ DEL then
            MAP := MAP ∪ {(p.a, p.d)}
            compareTrees (p.a, p.d, M', A', D')
            ADD := ADD- {p.a}, DEL := DEL- {p.d}
            removeMatches(Candidate, p) /* remove candidate matches that
               include p.a or p.d                                         */
        end
        ADD := ADD ∪ A', DEL := DEL ∪ D', MAP := MAP ∪ M'
    end
    findCandidate (ADD, DEL, Candidate, th_l, th_s)
until Candidate ≠ ∅ ;
interpret (MAP, ADD, DEL)
```

Function `compareTrees(`*L, R, M, D, A*`)`

```
/* align L and R's subtrees using the longest common subsequence
   algorithm based on their labels                                        */
MAP := alignLCS (L's first level subtrees, R's first level subtrees)
foreach l ∈ L's first-level subtrees do
    if l ∉ MAP.Left then DEL := DEL ∪ {l}
end
foreach r ∈ R's first-level subtrees do
    if r ∉ MAP.Right then ADD := ADD ∪ {r}
end
foreach (l,r) ∈ MAP do
    compareTree (l, r, MAP, DEL, ADD);
end
```

Function `findCandidate`(*A, D, Candidate,* th_l*,* th_s)

```
foreach a ∈ ADD do
    foreach d ∈ DEL do
        likeness := textSimilarity (a,d)
        if ((a.text.length > 128 or d.text.length > 128) and likeness > th_l) or
        (a.text.length < 128 and d.text.length < 128 and likeness > th_s) then
            Candidate := Candidate ∪ { (a,d,likeness) }
        end
    end
end
```

Once the initial ADD and DEL have been populated, the algorithm then tries to match nodes from ADD and DEL using a greedy version of a weighted bipartite graph matching algorithm. First, for each pair in the Cartesian product of ADD and DEL, we compute the pair's weight using the text similarity algorithm in UMLdiff [34], which computes how many common adjacent character pairs are contained in two compared strings. The weight calculation is based on the full content of the node's subtree. For example, when considering an `always` block node, the text of its block declaration and its body is used. If the similarity value is above a required threshold and the nodes are of the same syntactic type, such as an `always` block mapping to an `always` block, we add the pair to the set of potential matches, $Candidate$.

When computing text similarities, we use two different thresholds. For text that is less then 128 characters a lower threshold th_s is used, because small changes have a relatively larger effect on the similarity calculation. While most single line statements are kept under 128 characters, process blocks tend to be multi-line statements, requiring a larger threshold value to ensure a quality match.

Once all pairs in $\{ADD \times DEL\}$ have been evaluated, the potential match set $Candidate$ is sorted in descending order based on the pair's text similarity. Then we use a greedy algorithm to select a subset of $Candidate$. In each iteration, we take the highest weighted pair and add it to the set of matched nodes, MAP, and update $Candidate$ by removing all candidate matches that include either the selected pair's left or right hand side. The children of the matched pair are recursively compared to find any more additions, deletions, or matches. At the end of the iterations, ADD, DEL, and $Candidate$ are updated to account for newly matched nodes. This iteration continues until no new candidate matches are found. For each pair (a, d) in MAP, if the full text of a matches the full text of d exactly, they share the same parent, and their execution orders do not matter (i.e., `always`, `initial`, `generate`, `assign`, and `<=`), then the pair is removed from MAP and marked as unchanged.

Table 1 Change Types for Verilog Programs

Syntactic Element	Pattern	Description
Always	AL_ADD	Always block added
	AL_RMV	Always block removed
	AL_SE	Changes in the sensitivity list
Assignment Statement	ASG_ADD	Continuous assignment added
	ASG_CE	Continuous assignment changed
	ASG_RMV	Continuous assignment removed
Blocking Assignment	B_ADD	Blocking assignment added
	B_CE	Blocking assignment changed
	B_RMV	Blocking assignment removed
Non-Blocking Assignment	NB_ADD	Non-blocking assignment added
	NB_CE	Non-blocking assignment changed
	NB_RMV	Non-blocking assignment removed
If Statement	IF_ABR	Addition of else branch
	IF_APC	Addition of if branch
	IF_CC	Change of if condition expr
	IF_RBR	Removal of else branch
	IF_RMV	Removal of if branch
Switch Statement	SW_ABRP	Changes to switch hierarchy
	SW_CADD	Addition of a case branch
	SW_CRMV	Removal of a case branch
	SW_CHG	Changes to condition
Module Declaration	MD_CHG	Changes in port type/width
	MD_DNP	Different number of ports
Module Instantiation	MI_ADD	Module instantiation added
	MI_RMV	Module instantiation removed
	MI_DCP	Different ports values
	MI_DNP	Different number of ports
	MI_DTYP	Different types
Initialization	INIT_ADD	Initial block added
	INIT_RMV	Initial block removed
Parameter	PARAM_ADD	Parameter added
	PARAM_CHG	Parameter changed
	PARAM_RMV	Parameter removed
Register	RG_ADD	Register added
	RG_CHG	Register changed
	RG_RMV	Register removed
Wire	WR_ADD	Wire added
	WR_CHG	Wire changed
	WR_RMV	Wire removed
Pre-processor Directives	**Pattern**	**Description**
Define	DEFINE_ADD	DEFINE added
	DEFINE_CHG	DEFINE changed
	DEFINE_RMV	DEFINE removed
Ifdef	IFDEF_ADD	IFDEF added
	IFDEF_CHG	IFDEF changed
	IFDEF_RMV	IFDEF removed
Include	INC_ADD	Include added
	INC_RMV	Include removed
Generate	GEN_ADD	Generate block added
	GEN_RMV	Generate block removed
	GEN_CHG	Generate block changed
Others		
	NC	Formatting Changes

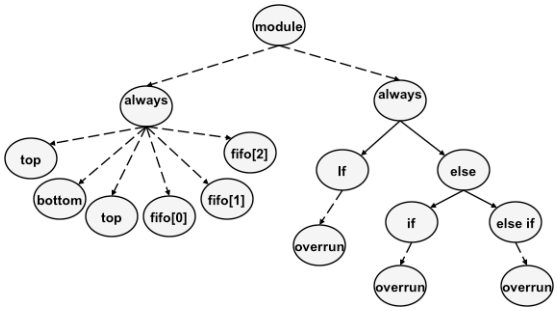

Fig. 3 AST of uart_rfifo.v from Figure 1

3.2 Change-Types for Verilog

In order to provide differencing results at a higher abstraction level than simply listing *ADD*, *DEL*, and *MAP*, we output syntactic differences in terms of change types. This classification can potentially help users understand the differences quickly by providing a set of categories that the hardware designer can easily identify with. Furthermore, change classification can enable quantitative and qualitative assessments of frequent change types in Verilog by providing a detailed uniform description of code changes.

The initial set of change types are motivated from Sudakrishnan's change types [30]. By manually inspecting all versions of OpenCores project UART16550 and the DRAM Memory Controller of the RAMP project (see Section 4), we created a new change type if the change did not fit within the classification list. The resulting list of change types is shown in Table 1.

Each of the major categories in the list has to do with a specific syntactic element in Verilog. For example, IF deals with `if` statement; MD and MI deal with module declarations and instantiations respectively; ASG focuses on assignment statements; AL focuses on `always` blocks, etc. Figure 6 shows an example of IF_CC and IF_RMV changes.

As a part of a post processing step, where Vdiff interprets the matching results between two abstract syntax trees as Verilog-specific change types, it refines the results in process interface declarations by extracting Boolean expressions from the AST nodes and checking their equivalence using a SAT solver. We used the SAT4J public java library, which takes Boolean formula in a conjunctive normal form (CNF) and proves whether there exists a set of inputs that can satisfy the formula [4]. This is similar to Person et al.'s differential symbolic execution technique [24] in that syntactic differencing is complemented by using a decision procedure for checking semantic equivalence. While differential symbolic execution techniques compute symbolic summaries at a method (or block) and check equivalence between two methods [24,19], Vdiff checks equivalence between sensitivity lists (i.e., Verilog's process interface descriptions written in Boolean logic) and does not perform extensive symbolic execution like Person et al.'s technique.

For example, the first `always` block sensitivity list in Figure 1 was reordered between versions. From a syntactic point of view, there has been a definite change to the sensitivity list; however, the change has no effect on the operation of the `always` block because the modified list is equivalent to the original list for every possible set of input signals. We currently focus on checking changes to an `always` block sensitivity list (AL_SE) to see if the original and modified lists are *Boolean equivalent*. In the future version of Vdiff, we plan to extend our SAT solver-based semantic comparison to include Boolean expressions in blocking and non-blocking assignments, continuous assignments, and IF conditions.

3.3 Vdiff Eclipse Plug-In

We implemented our differencing algorithm as an Eclipse plug-in. The plug-in is available for download [15]. Vdiff plug-in compares program revisions retrieved from a Subversion repository using the Subclipse interface [5]. Figure 2 shows the screen snapshot of our Vdiff plug-in. Its tree view visualizes AST matching results hierarchically; its text view presents textual differences between two program versions using the Eclipse *compare* plug-in and its console outputs change-type level differences with a pointer to word level differences. For example, changes to the sensitivity list are identified as textual differences in the side-by-side view, but they are reported as AL_SE: sensitivity list changes. As reordering input signals in the sensitivity list does not lead to any semantic differences, the change is marked as *Boolean equivalent*.

4 Quantitative Evaluation

Our evaluation addresses the following research questions:

- RQ1: What is the overall accuracy of Vdiff in computing change-type level differences?
- RQ2: How does Vdiff's AST matching algorithm compare to existing AST matching algorithms?
- RQ3: What is the impact of using similarity thresholds in matching AST nodes?
- RQ4: How well does Vdiff scale to large projects with a large amount of changed code?

Subject Programs. To evaluate Vdiff, we acquired data from two Verilog projects: UART16550 [2] and GateLib's DRAM controller project [3]. The UART16550 project contains the design for the core logic of a serial communication chip that provides communication capabilities with a modem or other external devices. We also analyzed the RAMP project's GateLib DRAM controller. RAMP is an infrastructure used to build simulators using FPGAs (field-programmable gate arrays). To be able to access memory uniformly independent of a chosen platform, GateLib's DRAM controller provides an abstract

Table 2 Subject Programs

	UART16550	GateLib
LOC	2095 to 3616	286 to 1843
# Files	8 to 12	1 to 5
# Check-ins	56	49
Avg. Modified Lines	42.12	27.98
Avg. Modified Files	1.92	1.35

interface which includes a standard DRAM interface, arbiter, asynchronous adapter and remote memory access.

To evaluate the accuracy of Vdiff output, we created an evaluation data set through manual inspection. We examined the individual output of svn diff on the same revision and manually classified them into change-types. Vdiff ran on the same version histories and produced change-type level differences. Running Vdiff took 0.080 second per revision on average (in comparison to 0.059 second on average when running GNU diff) on Intel Core 2 Duo Thinkpad 2 GHz with 1.96 GB of RAM running Windows XP. Vdiff's output was then compared to the manually created evaluation data set. In addition, to assess performance and scalability, we applied Vdiff to the revision history of nine open source projects using a machine configuration, an Intel Core i5 M 520, 2.40 GHz Dual Core processor with 2GB of RAM, running Windows 7 Professional 32-bit.

4.1 Precision and Recall

Suppose that V is a set of change-type level differences identified by Vdiff, and E is a set of manually identified change-type level differences. Precision and recall are defined as follows:

Precision: the percentage of Vdiff's change-type level differences that are correct, $\frac{|V \cap E|}{|V|}$

Recall: the percentage of correct change-type level differences that Vdiff finds, $\frac{|V \cap E|}{|E|}$.

Figure 4 shows the results on UART16550 project's 56 check-ins and GateLib project's 49 check-ins. Each row reports the number of revisions per file, the size of an evaluation data set (i.e., manually inspected change-types $|E|$), the number of change-type level differences reported by Vdiff ($|V|$), the number of *correct* differences reported by Vdiff ($|V \cap E|$), and precision and recall per file. Our evaluation shows that Vdiff is extremely accurate for most modules—its precision and recall are 97.5% and 97.7% on UART16550 and 96.2% and 96.9% on GateLib's DRAM controller.

The inability to match nodes due to low text similarity led to false positives (incorrect differences found by Vdiff, $V - E$) and false negatives (correct differences that Vdiff could not find, $E - V$). Figure 6 shows an example of both false positives and false negatives. In this example, three changes were made: (1) an extra condition (`rstate == sr_idle`) was added before setting `counter_b` (IF_CC), (2) the condition for decrementing `count_b` was modified by removing

File	Rev.	LOC (min:max)	Eval. $\|E\|$	Vdiff $\|V\|$	$\|V \cap E\|$	Prec. $\|V \cap E\|$ /$\|V\|$	Rec. $\|V \cap E\|$ /$\|E\|$
raminfr.v	3	95:111	3	3	3	100%	100%
timescale.v	3	3:64	3	3	3	100%	100%
uart_debug_if.v	6	98:126	9	9	9	100%	100%
uart_defines.v	10	177:247	25	25	25	100%	100%
uart_receiver.v	25	341:482	94	103	92	89.3%	97.9%
uart_regs.v	35	531:893	282	276	272	98.6%	96.5%
uart_rfifo.v	5	267:320	37	37	37	100%	100%
uart_sync_flops.v	2	122:122	2	2	2	100%	100%
uart_tfifo.v	3	227:243	3	3	3	100%	100%
uart_top.v	11	170:340	38	38	38	100%	100%
uart_transmitter.v	13	288:351	39	39	39	100%	100%
uart_wb.v	25	125:317	65	63	63	100%	96.9%
Total (UART)	141		600	601	586	97.5%	97.7%
DRAM.v	14	286:297	22	23	21	91.3%	95.5%
DRAMArbiter.v	15	286:429	214	224	214	95.5%	100.0%
DRAMArbiterInner.v	5	392:396	5	5	4	80.0%	80.0%
DRAMExaminer.v	29	180:450	249	244	238	97.5%	95.6%
DRAMRouter.v	6	397:397	7	6	5	83.3%	71.4%
Total (GateLib)	69		497	502	482	96.2%	96.9%
Total	210		1097	1103	1068	96.8%	97.3%

Fig. 4 Precision and recall of Vdiff on subject programs (th_s=0.65 and th_l=0.80)

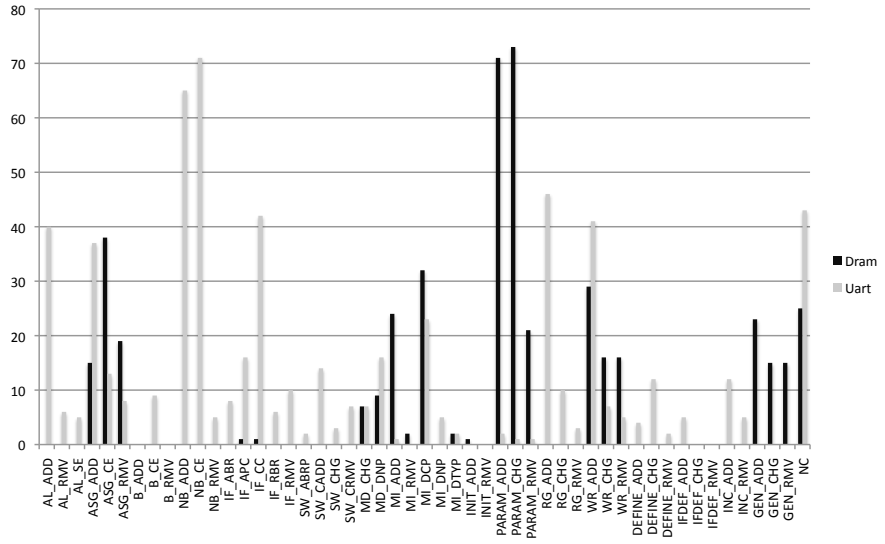

Fig. 5 Frequency of change-types

`counter_b != 8'hff` (IF_CC), and (3) the else block with the `rx_lsr_mask` condition was removed (IF_RBR). Since the text similarity algorithm used by Vdiff considers the first IF condition different enough from its original, the change is actually classified as a removal of an IF statement (IF_RMV) with an addition of a new IF statement (IF_APC). Thus, Vdiff reports two incorrect change-type level differences (IF_RMV, IF_APC) and misses three expected differences as a result (IF_CC, IF_CC, IF_RBR).

Expected Differences

(old program version)

```
- if (!srx_pad_i)   /* IF_CC */
-    counter_b <= #1 8'd191;
- else
-   if (counter_b != 8'b0 && counter_b != 8'hff)   /* IF_CC */
-   counter_b <= #1 counter_b - 8'd1;
-   else if (rx_lsr_mask)
-     counter_b <= #1 8'hff;   /* IF_RMV */
```

(new program version)

```
+ if (!srx_pad_i || rstate == sr_idle)   /* IF_CC */
+   counter_b <= #1 8'd191;
+ else
+    if (counter_b != 8'b0)   /* IF_CC*/
+      counter_b <= #1 counter_b - 8'd1;
```

Vdiff Outputs

(old program version)

```
- if (!srx_pad_i)   /* IF_RBR *:
-    counter_b <= #1 8'd191;
- else
-   if (counter_b != 8'b0 && counter_b != 8'hff)
-     counter_b <= #1 counter_b - 8'd1;
-   else if (rx_lsr_mask)
-     counter_b <= #1 8'hff;
```

(new program version)

```
+ if (!srx_pad_i || rstate == sr_idle)   /* IF_APC */
+    counter_b <= #1 8'd191;
+ else
+   if (counter_b != 8'b0)
+     counter_b <= #1 counter_b - 8'd1;
```

Fig. 6 Vdiff reported IF_APC and IF_RMV when two IF_CCs and one IF_RBR were expected. Code with red shade represents removal, code with gray shade represents modification, and code with blue shade represents addition.

To understand the types of changes common in Verilog, we plotted the distribution of identified change-types in Figure 5. The two projects we analyzed had very different characteristics: UART16550 had a significant number of core logic changes during its development, whereas GateLib's DRAM project evolved its abstract interface while hiding the actual implementation of the platform-specific DRAM implementation. In the UART16550 design, changes were frequently made to non-blocking assignments, registers, and `always` blocks. GateLib project had many changes to `generate` blocks and parameters. Ubiquitous changes observed across both projects were wire additions, changes to module instantiation ports, and changes to assignments.

We hypothesize that by producing accurate syntactic differences in terms of change-types, Verilog developers can better understand differences at a high level of abstraction. In addition to our user study participants described in Section 5, we demonstrated Vdiff to a few engineer with many years of experience in Verilog. One of the designers told us, *"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]."* We plan to study how engineers use Vdiff on their codebase, measure its accuracy with respect to the differences expected by the engineers, and improve Vdiff's algorithm based on their suggestions.

4.2 Comparison of AST Matching Algorithms

To assess the effectiveness of our weighted bipartite graph matching algorithm in matching AST nodes in the same level, we constructed two alternative algorithms by borrowing ideas from existing AST matching algorithms [12, 22].

1. Exact Matching: This is the most naïve version of AST matching algorithm that finds corresponding nodes in the same level using the exact same label. It has the same effect of using Neamtiu et al.'s AST matching algorithm [22] that traverses two trees in parallel and matches corresponding nodes by the same label in the same syntactic position in the trees.
2. In-Order Matching: This algorithm finds corresponding nodes in the same level in order—it starts by examining each node in the left tree in order and searching a node in the right tree with the highest similarity. This algorithm has the same effect of using the Cottrell et al.'s AST matching algorithm [12], which determines ordered correspondences between two sets of descendant nodes by considering nodes in the left tree in turn and finding the best corresponding node in the right tree using a linear search.
3. Greedy Weighted Bipartite Matching: Our algorithm finds corresponding nodes in the same level using a weighted bipartite graph matching algorithm [11].

Table 3 shows the comparison of the precision and recall of in-order matching algorithms (column 1 and column2) with our weighted bipartite matching, which relaxes the constraint of linear search to prevent early selection of a match that leads sub-optimal matching (column 3). As shown in Table 3, our algorithm improved the precision by 41.4% and the recall by 29.8% compared to the baseline (column 1) and improved the precision by 6.6% and the recall by 5.9% compared to an in-order matching based on similar labels (column 2). This evaluation of 1097 differences from 210 file revisions in two real world projects shows that *the ordering of code actually matters in practice* when it comes to computing differences between program versions.

Based on the anonymous reviewers' comments from ASE 2010, we compared Vdiff with the EMF configurable program differencing framework [1]

Table 3 Comparison between different algorithms for matching sibling nodes

Average	Label Matching	In-Order Matching	Weighted Bipartite
Precision	56.1%	90.9%	97.5%
Recall	67.9%	91.8%	97.7%

by adapting it to work for Verilog. We also tried to compare Vdiff with Sidiff [28] but Sidiff was not available for an extension to target Verilog. We mapped (1) modules in Verilog to classes in EMF, (2) `always` blocks and continuous assignments to operations, (3) wires, registers, and ports to fields, (4) and module instantiations to reference pointers in an EMF ecore model. On the same UART data set, the EMF Compare tool reported the 47.04% recall with the 80.84% precision because the EMF ecore modeling language could not model the implementation of `always` blocks including blocking and non-blocking statements.

4.3 Impact of Similarity Thresholds

Our algorithm uses th_s (threshold for short text) and th_l (threshold for long text) to determine the similarity between two AST subtrees. If the similarity is above an input threshold, then the difference will be classified as change; otherwise, they are considered an ADD or a DELETE. We assessed the impact of these similarity thresholds by incrementing th_s by 0.05, from 0.5 to 0.95, while setting th_l to its default value 0.80. We also incremented th_l by 0.05, from 0.5 to 0.95, while setting th_s to 0.65. Figures 7 and 8 show the resulting accuracy of varying these thresholds on the `uart_receiver.v` file during its entire revision history. The F-measure is also plotted to reason about precision and recall together: $\frac{2 \times Precision \times Recall}{(Precision + Recall)}$.

Precision generally increases as th_s increases due to more strict matching requirements. If th_s is too low, unrelated nodes are incorrectly matched and reported as changes instead of additions and deletions, adversely affecting accuracy. However, when th_s reaches around 0.95, its precision and recall measures decrease as the threshold requirement becomes too strict, and many unmatched nodes are considered additions and deletions instead of expected changes. The F-measure reaches the maximum when th_s is 0.65. Varying th_l follows a similar trend for similar reasons. However, matching large blocks requires a more strict threshold for correct matching to occur as illustrated by the increase in precision from 0.60 to 0.90. The F-measure reaches the maximum when th_l is around 0.8 and 0.85.

4.4 Performance

To measure the running time performance and scalability, we applied Vdiff to the entire revision history of nine open source hardware design projects

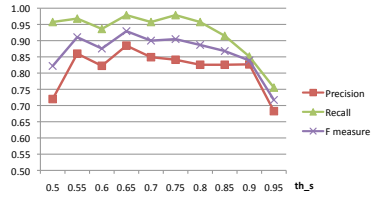

Fig. 7 Precision and recall when varying th_s while keeping th_l at 0.80

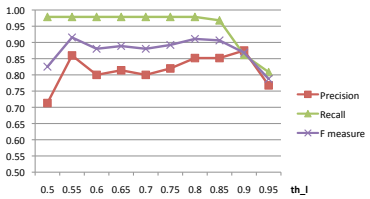

Fig. 8 Precision and recall when varying th_l while keeping th_s at 0.65

found in the OpenCores repository and the RAMP project repository.[1] These projects are Amber ARM-compatible core for 3-stage pipeline, 5-stage pipeline, and core system components; Ethernet MAC 10/100 Mbps; OpenMSC430 16-bit micro-controller; OpenRISC OR1200 processor; RAMP GATELIB DRAM controller; Secure Digital (SD) card host controller; and UART 16550 core. We selected these nine projects because they are generally larger than other projects in size. The size of the latest version of each project ranges from 1842 LOC to 31473 LOC. We measured the total number of added and deleted lines over all revisions per each file using its SVN history. The cumulative number of changed lines per each project during its evolution history ranges from 265 CLOC to 17288 CLOC (Changed LOC). For these input programs, running Vdiff on its entire revision history for all files takes only 0.08 minutes for Amber ARM-compatible core for 3-stage pipeline; 0.21 minutes for Amber for 5-stage pipeline; 0.24 minutes for Amber system; 1.54 minutes for Ethernet MAC 10/100 Mbps; 0.53 minutes for OpenMSC430 16-bit micro-controller; 1.62 minutes for OpenRISC 0R1200 processor; 0.28 minutes for RAMP GATELIB DRAM controller; 0.07 minutes for SD card host controller; and 0.77 minutes for UART 16550 core.

The running time of Vdiff depends on both the size of input program pairs and program differences. Thus Table 4 shows both the size of the latest program version in LOC (lines of code), the sum of program differences over all revisions in CLOC ($\sum$ Changed LOC), and the time taken to run Vdiff in milliseconds, and the total number of AST node additions or deletions over all revisions produced by Vdiff ($\sum$ AST edits).

Because most open source hardware design projects in the OpenCores repository are moderate size up to 30 KLOC, to measure scalability, we combined the evolution history of the above nine projects and measured the cumulative running time for processing the entire revision history of each additional file. Figures 9, 10, and 11 show the cumulative distribution of running time in milliseconds vs. the size of the latest version in LOC, the total size of all added and deleted lines, and the total size of Vdiff output in terms of AST node edits respectively.

[1] `http://opencores.org` and `http://ramp.eecs.berkeley.edu`

Table 4 Vdiff running time for nine open source subject programs

Program	Latest Size (LOC)	$\sum$ CLOC	Time (ms)	$\sum$ AST edits
`Amber ARM-compatible core (3-stage pipeline)`	6455	265	4994	59
`Amber ARM-compatible core (5-stage pipeline)`	10393	2142	12654	106
`Amber ARM-compatible core (system)`	5805	2154	14179	100
`Ethernet MAC 10/100 Mbps`	12320	17288	92889	597
`OpenMSC430 16-bit micro-controller`	8661	6040	32141	638
`OpenRISC OR1200 processor`	31473	16409	97137	484
`RAMP GATELIB DRAM controller`	1842	1948	17000	244
`Secure Digital (SD) card host controller`	2924	4780	4391	50
`UART 16550 core`	3616	2991	45853	448

The above projects are all downloaded from the OpenCores repository at `http://opencores.org`.

Vdiff running time increases linearly as we increase the amount of added and deleted code, as shown in Figures 11 and 10. The running time also increases roughly linearly, as we increase the input program size, as shown in Figure 9. In summary, Vdiff processes the entire version history of 54017 modified lines from the nine projects and produces 2726 AST node additions/deletions all under 5.35 minutes.

5 User Study

To assess the utility of Vdiff during peer code reviews, we conducted a user study with eight hardware design experts from a large multinational semiconductor and processor company. The goal of this user study is to understand how hardware designers identify and classify Verilog program differences during peer code reviews and to assess how the experts' change classification matches the output of Vdiff. Furthermore, our goal is to understand and compare the strengths and limitations of Vdiff against Tkdiff, a tool that visualizes textual program differences in color in a side by side view. Tkdiff is widely used among hardware design engineers in the same company [6].

We recruited participants by sending email invitations to engineers and managers in the RTL (register-transfer level) design group in the company. Eight engineers responded to our study invitation and participated in the study. Table 5 summarizes the profile of study participants. The participants had three to thirty years of experience in hardware design and all participants used a textual program differencing tool, *diff*, and *diff*-based version control systems. Six out of eight participants have participated in peer code reviews before. In terms of their role in the company, P1, P2, P5, and P7 are RTL

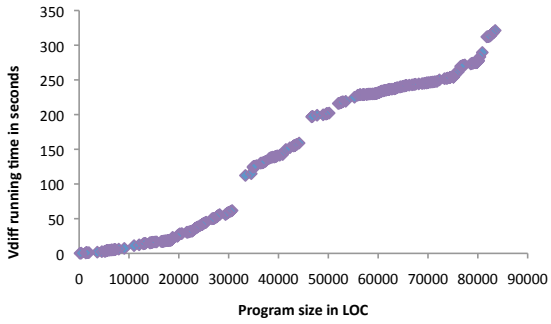

Fig. 9 Program size in LOC vs. Vdiff running time in seconds

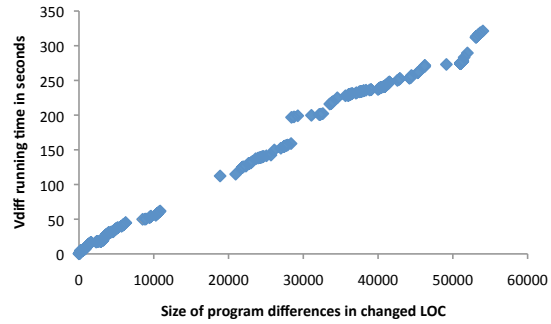

Fig. 10 The size of program differences in changed LOC vs. Vdiff running time in seconds

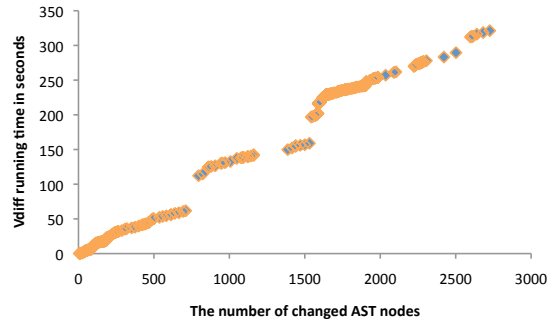

Fig. 11 Vdiff output size, i.e., the number of AST node additions and deletions vs. Vdiff running time in seconds

designers; P3 is a verification team lead; P4 and P8 are design team leads; and P6 is an RTL team manager.

The study participants were given two code review tasks. For each task, the participants inspected changes between the old and new version of a Verilog program and classified program differences explicitly as semantic or non-semantic differences. The following paragraph shows the instructions provided to the participants.

Table 5 Background of User Study Participants

Background Questions

Q1	How many years of experience do you have in the hardware design industry?
Q2	What is your job title? (optional)
Q3	How many years of experience do you have for the following hardware description languages (HDLs)? a. Verilog? b. SystemVerilog? c. VHDL?
Q4	Have you used the diff or svn diff tool before?
Q5	Have you used an integrated development environment (IDE) like Eclipse to develop with HDLs before?
Q6	Have you participated in code reviews before?
Q7	Which version control systems have you used (such as SVN, bitkeeper, CVS, Clearcase)?

Answers from Participants

	Q1 (Years)	Q2 (Job Title)	Q3 Verilog	System Verilog	VHDL
P1	8	Staff design engineer	8	2	0.5
P2	30+	Principal design engineer	0.5	0	0
P3	14	Principal member of technical staff	14	10	2
P4	22	Consultant design engineer	6	0	9
P5	20	Consultant design engineer	18	0	0
P6	19	Principal design engineer	18	1	0
P7	15	Consultant design engineer	15	0	4
P8	3	Senior design engineer	5	1	7

	Q4 (diff)	Q5 (IDE)	Q6 (Code Review)	Q7 (Version control systems)
P1	Yes	No	Yes	SVN, CVS, Synchromicity
P2	Yes	No	No	CVS, SVN
P3	Yes	No	Yes	SVN, Clearcase, Synchronocity
P4	Yes	No	Yes	SVN, RCS, CVS
P5	Yes	No	Yes	SVN, CVS, Synchronocity
P6	Yes	No	Yes	SVN, RCS, CVS
P7	Yes	No	Yes	CVS, SVN, Perforce
P8	Yes	Yes	No	SVN, Mercurial

"The goal of this user study is to understand how hardware design engineers interpret program differences between two versions of a Verilog file. After you answer short questions about your background, you will be presented with two sets of Verilog examples, each with an "original" version and a "changed" version. Please identify each change you recognize and classify it (such as "condition of IF statement changed"). Please also note whether the changes that you have recognized are semantic changes or non-semantic changes."

The code examples for the user study were taken from the `uart_receiver.v` design, a part of the 16550 UART Open Cores (universal asynchronous receiver/ transmitter) project. The SVN revisions 44 and 45 were used to create the examples. These examples were selected by the first two authors, who have eleven and eight years of hardware design experience in industry respectively, because these tasks involve fairly complex program differences in Verilog. These tasks also require reviewing both semantic and non-semantic changes, which are very common in Verilog programs, emulating realistic code review tasks in hardware design firms. A few edits were made to condense the

code to fit in two pages and to partition the changes from multiple version of the file into two sets of differences. This simplification was done not to favor our Vdiff, which handles very large programs fast as shown in Section 4.4, but to make the code review tasks completed within 30 minutes, which we ask from the participants during our recruitment. Figures 12 and 13 show the Verilog programs used for the code review tasks. The Verilog programs provided to the user had syntax highlighting but did not have any change annotations such as '① renaming `rf_push_pulse` to `pulse`' shown in the figures. These change annotations are inserted later for presentation purposes for this article. None of the participants have experience with developing or extending the UART Open Cores project.

For the first code review task, the design in Figure 12 contains a simple set of counters (`b` and `t`) and a `pulse` signal. The reset signal `wb_rst_i` sets the counters and clears `pulse`. When enabled, the counters decrement. While it contains many changes common during the evolution of Verilog design, some of the changes have no effect on the semantics. For example, changes ② and ③ reorganize the code; thus, the functionality is not affected by the changes. Other changes that modify the Boolean algebra such as ④ and ⑤ directly affect the semantics of the hardware design. Finally, changes such as ①, ⑥, and ⑦ reflect signal name changes. In terms of semantics, these renamed signals may affect other areas of the design. Furthermore, these renamings many not be obvious to the code reviewer because the old and new signal names are very similar.

For the second review task, the design in Figure 13 contains a state machine with an appropriate reset and the same pulse design as in Figure 12. The differences noted in Figure 13 again contain both semantic and non-semantic differences. ① and ③ reflect code reordering with no semantic change. Similarly, changes ⑤ and ⑦ reflect an ordering change inside of `always` blocks without semantic differences. Changes ② and ④ modify an assignment and add a new signal. Moreover, change ⑧ modifies an assignment. Finally, change ⑨ shows a new constant definition and its use.

After each code review task, the participants were presented with two sets of program differencing outputs produced by Vdiff and Tkdiff respectively. They were then asked to assess how the tools' outputs match their change classifications. Each study consists of an explanation of a study procedure (5 minutes), the first code review task (10 minutes), the second code review task (10 minutes), assessment of Vdiff and Tkdiff outputs on the same code (5 minutes). Each user study session was audio-recorded.

Table 6 summarizes each participant's classification of program differences. It also shows Vdiff and Tkdiff's outputs and whether each change is a semantic difference or a non-semantic difference. $\surd$ represents a change that a participant explicitly noted as a semantic difference. ∇ represents a change that participants explicitly noted as a non-semantic difference. Empty cells indicate that the participant did not make any comment about the change. We computed an inter-rator agreement score by comparing each participant's classification against other participants. The inter-rator agreement score for

(old program version)

```
module uart_receiver (clk, wb_rst_i,
 rf_push, rf_pop, srx_pad_i, enable,
 // inputs
 counter_t, rf_count, rstate,
  ①rf_push_pulse );
 // outputs
  // Determine counter B value
  always @ (posedge clk or posedge
  wb_rst_i)
    begin
      if (wb_rst_i)
        ⑥counter_b <= 8'd159;
      else
        if (srx_pad_i)
          ⑥counter_b <= brc_value;
  // character time length-1
        else
          if (enable & ⑥counter_b !=
    8'b0)
  // only work on enable times
  // break not reached.
            ⑥counter_b <= counter_b -1;
  // decrement break counter
    end // always of break condition
  // detection
  // Timeout condition detection
  ②reg [9:0] counter_t;
  // counts the timeout condition clocks
  // Determine counter T value
  ③always @ (posedge clk or posedge
  wb_rst_i)
    begin
      if (wb_rst_i)
        counter_t <= 10'd639;
    // 10 bits for the default 8N1
      else
        if ④(rf_push_pulse || rf_pop
  || rf_count == 0)
        // cntr reset if RX FIFO @
        // trigger level/accessed/empty
          counter_t <= toc_value;
        else
          if ( ⑤enable && counter_t !=
   10'b0)
  // we don't want to underflow
          counter_t <= counter_t-1;
    end
  always @ (posedge clk or posedge
  wb_rst_i)
    begin
      if (wb_rst_i)
        rf_push_q <= 0;
      else
        rf_push_q <= rf_push;
    end
  assign ⑦rf_push_pulse = rf_push
  & ~rf_push_q;
endmodule
```

(new program version)

```
module uart_receiver (clk, wb_rst_i,
 rf_push, rf_pop, srx_pad_i, enable,
 // inputs
 counter_t, rf_count, rstate, ①pulse );
 // outputs
  // Timeout condition detection
  ②reg [9:0] counter_t;
  // counts the timeout condition clocks
  // Determine counter T value
  ③always @ (posedge clk or posedge
  wb_rst_i)
    begin
      if (wb_rst_i)
        counter_t <= 10'd639;
      // 10 bits for the default 8N1
      else
        if ④(rf_pop)
  // cntr reset if RX FIFO @
  // trigger level/accessed/empty
          counter_t <= toc_value;
        else
          if ( ⑤~enable && counter_t !=
    10'b0)
  // we don't want to underflow
            counter_t <= counter_t - 1;
    end
  // Determine counter B value
  always @ (posedge clk or posedge
  wb_rst_i)
    begin
      if (wb_rst_i)
        ⑥rcounter_b <= 8'd159;
      else
        if (srx_pad_i)
          ⑥rcounter_b <= brc_value;
        // character time length - 1
        else
          if (enable & ⑥ rcounter_b !=
          8'b0)
        // only work on enable times
        // break not reached.
            ⑥rcounter_b <= rcounter_b -1;
  // decrement break counter
    end // always of break condition
        // detection
  always @ (posedge clk or posedge
  wb_rst_i)
    begin
      if(wb_rst_i)
        rf_push_q <= 0;
      else
        rf_push_q <= rf_push;
    end
  assign ⑦pulse = rf_push
    & ~rf_push_q;
endmodule
```

Fig. 12 A Verilog program used for the code review task #1: an `always` block ③ and a register allocation ② are rearranged. `IF` conditions in ④ and ⑤ are modified. `counter_b` in ⑥ is renamed to `rcounter_b`. Variable `rf_push_pulse` is renamed to `pulse` in ① and ⑦.

(old program version)

```
module uart_receiver (clk, wb_rst_i, lcr,
  rf_pop, srx_pad_i, enable,
  // inputs
  counter_t, rf_count, rf_overrun,
  rstate, rf_push_pulse);
  // outputs
  always @ (posedge clk or posedge
  wb_rst_i)
    begin
      if (wb_rst_i)
        begin
          ② ④
          rstate          <= 1'b0;
          rcounter16      <= 0;
          rbit_counter    <= 0;
          ⑤rshift           <= 0;
          rf_push         <= 1'b0;
          ⑥rf_data_in       <= 0;
        end
      else
        if (enable)
          begin
            case (rstate)
              sr_idle : begin
                ⑧rcounter16 <= 4'b00000;
                ⑦rf_push    <= 1'b0;
                if (srx_pad_i==1'b0 &
                    ~break_error)
                  begin
                    rstate <= sr_rec_start;
                  end
              end
              sr_rec_prepare : begin
                case ⑨(lcr[1:0])
              // number of bits in a word
                  ...
              endcase
              rcounter16 <=
                rcounter16_minus_1;
              end
            endcase
          end
    end
  ③always @ (posedge clk or posedge
  wb_rst_i)
    begin
      if (wb_rst_i)
        rf_push_q <= 0;
      else
        rf_push_q <= rf_push;
    end
  ①assign rf_push_pulse = rf_push &
    ~rf_push_q;
endmodule
```

(new program version)

```
module uart_receiver (clk, wb_rst_i, lcr,
  rf_pop, srx_pad_i, enable,
  // inputs
  counter_t, rf_count, rf_overrun,
  rstate, rf_push_pulse);
  // outputs
  ①assign rf_push_pulse = rf_push &
    ~rf_push_q;
  ②assign sr_idle       = 1'b0;
  ③always @ (posedge clk or posedge
    wb_rst_i)
    begin
      if (wb_rst_i)
        rf_push_q <= 0;
      else
        rf_push_q <= rf_push;
    end
  always @ (posedge clk or posedge
    wb_rst_i)
    begin
      if (wb_rst_i)
        begin
          ④rstate          <= sr_idle;
          rcounter16      <= 0;
          rbit_counter    <= 0;
          ⑤rf_push         <= 1'b0;
          rshift          <= 0;
          ⑥
        end
      else
        if (enable)
          begin
            case (rstate)
              sr_idle : begin
                ⑦rf_push    <= 1'b0;
                ⑧rcounter16 <= 4'b1110;
                if (srx_pad_i==1'b0 &
                  ~break_error)
                  begin
                    rstate <= sr_rec_start;
                  end
              end
              sr_rec_prepare : begin
                case (lcr ⑨[/*[1:0]*/
                  `UART_LC_BITS] )
              // number of bits in a word
                ...
    endcase
                rcounter16 <=
                  rcounter16_minus_1;
              end
            endcase
          end
    end
endmodule
```

Fig. 13 A Verilog program used for the code review task #2: `assign` in ① and an `always` block in ③ are moved. A constant is replaced with a new variable ② and ④. Non blocking statements are reordered ⑤ and ⑦ and dead code is removed ⑥. A non-blocking statement had a semantic change ⑧. A new `define` statement is added.

semantic differences is the percentage of cells marked as $\surd$ out of all cells for the columns representing semantic differences (e.g., Columns ④ and ⑤ for the first task). Similarly, the inter-rator agreement score for non-semantic differences is the percentage of cells marked as ∇ out of all cells for the columns representing non-semantic differences (e.g., Columns ⑥, ⑦, ①, ②, and ③ for the first task). We also computed the agreement between the participants' change classification and Vdiff and Tkdiff outputs respectively.

The inter-rator agreement about semantic changes is 93.75% for the first task and 85.71% for the second task, indicating that designers could miss semantic changes and they do not necessarily converge on semantic changes. The inter-rator agreement about non-semantic differences is 71.42% and 66.66% respectively. These numbers are conservative lower-bound estimates as the designers did not enumerate all non-semantic changes even though they were requested to articulate all semantic and non-semantic differences that they recognize.

For both tasks, because Vdiff could detect register allocations and process blocks despite two `always` blocks having identical interfaces, the participants' change classifications were better aligned with Vdiff than Tkdiff. The agreement scores between Vdiff and participants are 45.67% and 38.88% for the first and second tasks respectively. There are two reasons why the agreement scores are low: (1) Vdiff found a few false positives because it could not detect renamed signals or a constant replacement with a new signal declaration. Such refactoring were responsible for four textual differences in the first task and two textual differences in the second task. (2) In case of non-semantic differences, the participants did not explicitly articulate all differences. Overall, it is noteworthy that Vdiff's output is better aligned with the experts' classification of Verilog program changes than Tkdiff.

When presented with the results of Vdiff and Tkdiff, the participants told us that Tkdiff reports too many non-semantic changes and it does poor job of recognizing re-arranged, semantically equivalent code. The following paragraphs lists some quotes from the participants regarding Tkdiff.

'Tkdiff is not smart enough to compare those two moved blocks.'

'It does not understand syntactic units of code...' *'I use Tkdiff, and it is a cave man's solution. I like its coloring scheme, but it is very poor at recognizing blocks that have been moved.'*

The participants reported that Vdiff does a good job at suppressing non-semantic differences, making it easier for them to filter out non-semantic differences. Verilog specific change-types in Vdiff help them to grasp a high level structure of design changes.

'The biggest problem that you are attacking (in Vdiff) is to ignore rearranged code, (in other words) logically equivalent code that diff cannot recognize ...'

'Wow, gotcha.. I think a nice thing about it is that this gives you a very high level structure of code... If I am looking at the code that I am not familiar with, this (Vdiff) gives you a high level structure of the design change, before

getting down into details.'

'It ignores things that you don't really care about, which is, this block getting re-positioned differently. That's not something material. Vdiff gives you material changes, logical changes that you as a designer cares about.'

'Yeah, I think this (Vdiff) is extremely useful. Tkdiff shows too much changes, but this tool would put me exactly at the material changes that I care about.'

Though most participants did not use integrated development environments before, it was surprising to see their openness to a new way of comparing a Verilog program—a common task that they do daily. Several users described a few painful techniques they have adopted to compare Verilog programs, such as manually editing the changed file to line up code blocks prior to running Tkdiff. After seeing Vdiff at the end of the session, the majority of the participants thought it would be useful and several of them asked how they can start using Vdiff today. A couple of users made a distinction in the type of tasks that they would use Vdiff for. They said they may not use Vdiff for comparing their own code, but would use it to comprehend changes made by others or code that they are not already familiar with.

Several participants suggested ideas for improving Vdiff. They wanted to see changed blocks highlighted in color, similar to Tkdiff, to make visual scanning easier. They also thought it would be nice to see more context lines around the change code—double-clicking on the change currently shows only a narrow view of surrounding code. Finally, they wanted to see program differences in conjunction with related signals, for example, tracing modified signals to their source and destination signals.

6 Discussion

This section discusses our Vdiff algorithms' limitations, threats to validity, and extensions necessary for applying Vdiff to other hardware description languages.

Limitations. Though we use two different thresholds, our algorithm is still sensitive to subtle changes to variable names or IF-conditions and requires careful tuning of similarity thresholds. Further investigation of different name similarity measures such as n-gram based matching [14] is required. Renaming wires, registers and modules often causes cascading false positives and false negatives by incorrectly matching AST nodes at a top-level. Renaming detection techniques [33, 20] could be used to overcome this limitation. The current algorithm cannot recover from mismatches at a top level as it matches parent nodes before matching their descendants. The equivalence check using a SAT solver is currently limited to only sensitivity lists due to VEditor's coarse-grained parsing algorithm, and we plan to extend this check to all types of boolean expressions. VEditor struggled with parsing pre-processor directives; consequently, we worked around IF-DEFs by creating a version where the IF branch is true and another version where the ELSE branch is true. We

Code Review Task 1. Figure 12

	Vdiff Change Types					
	④ IF_CC	⑤ IF_CC	⑥ NB_CE	⑥ IF_CC	⑥ NB_CE	⑦ASG_CE
Semantic?	Yes	Yes	No	No	No	No
Tkdiff	Yes	Yes	Yes	Yes	Yes	Yes
Vdiff	Yes	Yes	Yes	Yes	Yes	Yes
P1	√	√				∇
P2	√	√	∇	∇	∇	
P3	√	√	∇	∇	∇	
P4	√					
P5	√	√	∇	∇	∇	∇
P6	√	√	∇	∇	∇	∇
P7	√	√	∇	∇	∇	∇
P8	√	√	∇	∇	∇	∇
	Change Types Found by Users					
	① MOVE	② MOVE	③ MOVE			
Semantic?	No	No	No			
Tkdiff	Yes	Yes	Yes			
Vdiff	No	No	No			
P1	∇					
P2	∇	∇	∇			
P3	∇		∇			
P4	∇	∇				
P5		∇	∇			
P6	∇		∇			
P7		∇	∇			
P8	∇	∇	∇			

Inter-rator Agreement about Semantic Changes: 93.75%
Inter-rator Agreement about Non-semantic Changes: 71.42%*
Agreement between Vdiff and Participants: 45.67%*
Agreement between Tkdiff and Participants: 20.98%*

Code Review Task 2. Figure 13

	Vdiff Change Types				
	② ASG_ADD	⑥NB_RMV	④NB_CE	⑨SW_CADD	⑧NB_CE
Semantic?	No	No	No	Yes	Yes
Vdiff	Yes	Yes	Yes	Yes	Yes
Tkdiff	Yes	Yes	Yes	Yes	Yes
P1	∇		∇	√	√
P2	∇			√	√
P3	∇	∇	∇	√	√
P4	∇		∇	∇	√
P5	∇		∇		√
P6	(unavailable)				
P7	∇	∇	∇	√	√
P8	∇	∇	∇	√	√
	Change Types Found by Users				
	① MOVE	③ MOVE	⑤ MOVE	⑦MOVE	
Semantic?	No	No	No	No	
Vdiff	No	No	No	No	
Tkdiff	Yes	Yes	Yes	Yes	
P1	∇				
P2	∇		∇		
P3	∇	∇	∇	∇	
P4					
P5					
P6					
P7	∇	∇	∇	∇	
P8	∇	∇	∇	∇	

Inter-rator Agreement about Semantic Changes: 85.71%
Inter-rator Agreement about Non-semantic Changes: 66.66%*
Agreement between Vdiff and Participants: 38.88%*
Agreement between Tkdiff and Participants: 16.66%*

Table 6 √ indicates a change that participants explicitly noted as a semantic difference and ∇ represents a change that participants explicitly noted as a non-semantic difference. * The agreement scores are conservative lower bound estimates because most participants did not explicitly enumerate *all* non-semantic differences.

first computed differences for these two versions separately and later merged the results to help programmers understand syntactic differences under two possible circumstances. Our results on precision, recall and frequent change-types are limited to UART and GateLib and do not necessarily generalize to other projects. In addition, the construction and manual identification of change-types are subject to experimenter bias as they are done by the first two authors of this paper, who have eleven and eight years of hardware design experience in industry respectively.

Application of Vdiff to Other HDLs. While Verilog is the most widely used HDL, two other HDLs are also prevalently used: SystemVerilog and VHDL. SystemVerilog [27] extends the Verilog-2005 standard to include several features commonly found in modern object oriented programming languages: multi-dimensional arrays, `enum` data types, `struct`, `union`, strings, classes with inheritance, assertions, and synchronization primitives. Many of these features in SystemVerilog cannot be directly mapped to hardware circuitry but could be used for verification and simulation. VHDL [9] was initially developed in the 1980s, around the same time Verilog was created, and it has features similar to Ada. Vdiff could be easily extended to other HDLs by plugging in a different parser and handling new change types such as changes to `struct` or `enum` in SystemVerilog.

7 Related Work

Matching corresponding code elements between two program versions is a fundamental building block for version merging, regression testing selection and prioritization, and profile propagation. Existing differencing techniques often match code elements at a particular granularity based on closeness in name and structure, such as: (1) lines and tokens [18,26,10], (2) abstract syntax tree nodes [12,13,16,22,25,35], (3) control flow graph nodes [8], etc. In the context of hardware development with HDLs, the state-of-the-practice in comparing two versions of Verilog program is to use GNU *diff* [21] or a graphic front end to *diff*, which provides a side-by-side visualization that highlights deleted or added lines with different colors and provides navigation capability to review particular differences [6]. These line-based differencing tools use the longest common subsequence algorithm that aligns program lines in sequence [18].

For software version merging, Yang [35] developed an AST differencing algorithm. Given a pair of functions (f_T, f_R), the algorithm creates two abstract syntax trees T and R and attempts to match the two tree roots. Once the two roots match, the algorithm aligns T's subtrees $t_1, t_2, ..., t_i$ and R's subtrees $r_1, r_2, ...r_j$ using the LCS algorithm and maps subtrees recursively. This type of tree matching respects the parent-child relationship as well as the order between sibling nodes, but is very sensitive to changes in nested blocks and control structures because tree roots must be matched for every level. For dynamic software updating, Neamtiu et al. [22] built an AST-based algorithm

that tracks simple changes to variables, types, and functions. Neamtiu's algorithm assumes that function names are relatively stable over time. It traverses two ASTs in parallel, matches the ASTs of functions with the same name, and incrementally adds one-to-one mappings, as long as the ASTs have the same shape. In contrast to Yang's algorithm, it cannot compare structurally different ASTs. Cottrell et al.'s Breakaway [12] automatically identifies detailed structural correspondences between two abstract syntax trees to help programmers generalize two pieces of similar code. Its two-pass greedy algorithm is applied to ordered child list properties (e.g., statements in a block), and then to unordered nodes (e.g., method declarations). Their subsequent work, Jigsaw [13] improves Breakaway's AST matching algorithm. Jigsaw is different from ours in that Jigsaw leverages seed AST matches given by the developer in the context of copy and paste in an Eclipse IDE and also employs customized matching heuristics for different AST node types such as variable declarations, conditionals, loops, expressions, etc. It is not possible to compare our algorithm directly with Jigsaw because it uses semantically-based heuristics for different Java AST node types and the details are not provided in their paper.

Change Distiller [16] takes two abstract syntax trees as input and computes basic tree edit operations such as *insert, delete, move* or *update* of tree nodes. It uses *bi-gram string similarity* to match source code statements such as method invocations, and *subtree similarity* to match source code structures such as if-statements. After identifying tree edit operations, Change Distiller maps each tree-edit to an atomic AST-level change-type such as *parameter ordering change*. Vdiff uses an approach similar to Change Distiller, in that we identify similar subtrees by computing similarity measures and find the best matching among similar subtrees by selecting matches with the highest similarity one at a time. In addition, we also report AST-level matching results in terms of Verilog specific change-types. Raghavan et al.'s Dex [25] compares two ordered ASTs using both top-down matching and bottom-up matching. This algorithm gives preferences to AST node matches in the same level that do not results in moving or reordering nodes. Dex defines edit cost using fixed numbers instead of similarity between AST node labels. Yu et al.'s MCT detects meaningful program differences by leveraging programmer-provided annotations. It transforms input programs into a normalized form and remove clones across different normalized programs to detect non-trivial, relevant differences [36]. On the other hand, Vdiff does not require annotations.

The main difference between our AST comparison algorithm and existing AST matching algorithms is that our algorithm identifies syntactic differences robustly, even when multiple AST nodes have similar labels and when they are reordered.

In addition to these, several differencing algorithms compare model elements [34, 23, 29]. For example, UMLdiff [34] matches methods and classes between two program versions based on their name. However, these techniques assume that no code elements share the same name in a program and thus use name similarity to produce one-to-one code element matches. Our algo-

rithm differs from these by not relying on one-to-one matching based on name similarity.

As different language semantics lead to different program differencing requirements, some have developed a general, meta-model based, configurable program differencing framework [28,1]. For example, SiDiff [28,32] allows tool developers to configure various matching algorithms such as identity-based matching, structure-based matching, and signature-based matching by defining how different types of elements need to be compared and by defining the weights for computing an overall similarity measure.

Sudakrishnan et al. [30] studied the types of bugs that occur in Verilog and compared those findings to a similar study in Java. They presented a categorization of change-types that caused bugs and how often they occurred, and found that the most common bug pattern was changes to assignment statements and if-statements. In our work, we extended Sudakrishnan's change-types by adding 25 change-types to comprehensively describe code changes in two open source Verilog projects that we studied. While Sudakrishnan's change-type analysis is largely manual, our program differencing tool automatically identifies change-type level differences between two program versions.

8 Conclusion

Most program differencing algorithms implicitly assume sequential ordering between code elements or assume that code elements can be matched based on their unique names regardless of their positions, such as reordered Java methods. This limitation leads to poor accuracy when these techniques are applied to languages such as Verilog, where it is common to use non-blocking statements and there is a lack of unique identifiers for process blocks. This paper presented a position-independent AST matching algorithm that is robust to reordering of code elements even when their labels are not unique. Based on this algorithm, we developed Vdiff, a program differencing tool for Verilog. Our evaluation shows that Vdiff is accurate with a precision of 96.8% and a recall of 97.3% when using manually classified differences as a basis of evaluation. Our user study with eight hardware design experts shows that the experts' change classification is better aligned with Vdiff than Tkdiff and that Vdiff has potential to improve hardware designer productivity during peer code reviews.

Acknowledgements We thank Greg Gibeling and Derek Chiou for providing accesses to the RAMP repository and Adnan Aziz and anonymous reviewers for their detailed comments on our draft.

References

1. *Eclipse EMF Compare Project description: http://www.eclipse.org/emft/projects/compare.*
2. *Opencore.* http://opencores.org.

3. *Ramp*. http://ramp.eecs.berkeley.edu.
4. *Sat4J*. http://www.sat4j.org/.
5. *Subclipse*. http://subclipse.tigris.org.
6. *Tkdiff*. http://sourceforge/projects/tkdiff.
7. *Veditor*. http://veditor.sourceforge.net.
8. T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *ASE '04*, pages 2–13, Washington, DC, USA, 2004. IEEE Computer Society.
9. P. J. Ashenden. Vhdl-200x: The next revision. *IEEE Design and Test of Computers*, 20(3):112–113, 2003.
10. G. Canfora, L. Cerulo, and M. D. Penta. Tracking your changes: A language-independent approach. *IEEE Software*, 26:50–57, 2009.
11. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill Science/Engineering/Math, 2001.
12. R. Cottrell, J. J. C. Chang, R. J. Walker, and J. Denzinger. Determining detailed structural correspondence for generalization tasks. In *ESEC-FSE '07*, pages 165–174, New York, NY, USA, 2007. ACM.
13. R. Cottrell, R. J. Walker, and J. Denzinger. Semi-automating small-scale source code reuse via structural correspondence. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 214–225, New York, NY, USA, 2008. ACM.
14. L. R. Dice. Measures of the amount of ecologic association between species. *Ecology*, 26(3):297–302, 1945.
15. A. Duley, C. Spandikow, and M. Kim. Vdiff download, `http://users.ece.utexas.edu/~miryung/software/Vdiff.html`.
16. B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall. Change distilling—tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):18, November 2007.
17. V. Gupta and V. Pratt. Gates accept concurrent behavior. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:62–71, 1993.
18. J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, 1977.
19. S. K. Lahiri, K. Vaswani, and C. A. R. Hoare. Differential static analysis: opportunities, applications, and challenges. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pages 201–204, New York, NY, USA, 2010. ACM.
20. G. Malpohl, J. J. Hunt, and W. F. Tichy. Renaming detection. *Automated Software Engineering*, 10(2):183–202, 2000.
21. E. W. Myers. An o(nd) difference algorithm and its variations. *Algorithmica*, 1:251–266, 1986.
22. I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. In *MSR'05*, pages 2–6, 2005.
23. D. Ohst, M. Welle, and U. Kelter. Difference tools for analysis and design documents. In *ICSM '03*, page 13, Washington, DC, USA, 2003. IEEE Computer Society.
24. S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *SIGSOFT '08/FSE-16*, pages 226–237, New York, NY, USA, 2008. ACM. differential symbolic execution.
25. S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine. Dex: A semantic-graph differencing tool for studying changes in large code bases. In *ICSM '04*, pages 188–197, Washington, DC, USA, 2004. IEEE Computer Society.
26. S. P. Reiss. Tracking source locations. In *ICSE '08*, pages 11–20, New York, NY, USA, 2008. ACM.
27. D. I. Rich. The evolution of systemverilog. *IEEE Design and Test of Computers*, 20(4):82–84, 2003.
28. M. Schmidt and T. Gloetzner. Constructing difference tools for models using the sidiff framework. In *ICSE Companion '08*, pages 947–948, New York, NY, USA, 2008. ACM.
29. M. Soto and J. Münch. Process model difference analysis for supporting process evolution. *Lecture Notes in Computer Science, Springer Berlin*, Volume 4257/2006:123–134, 2006.

30. S. Sudakrishnan, J. Madhavan, E. J. Whitehead, Jr., and J. Renau. Understanding bug fix patterns in verilog. In *MSR '08*, pages 39–42, New York, NY, USA, 2008. ACM.
31. D. Thomas and P. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 2002.
32. C. Treude, S. Berlik, S. Wenzel, and U. Kelter. Difference computation of large models. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 295–304, New York, NY, USA, 2007. ACM.
33. P. Weißgerber and S. Diehl. Identifying refactorings from source-code changes. In *ASE '06*, pages 231–240, Washington, DC, USA, 2006. IEEE Computer Society.
34. Z. Xing and E. Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *ASE '05*, pages 54–65, New York, NY, USA, 2005. ACM.
35. W. Yang. Identifying syntactic differences between two programs. *Software – Practice & Experience*, 21(7):739–755, 1991.
36. Y. Yu, T. T. Tun, and B. Nuseibeh. Specifying and detecting meaningful changes in programs. In *ASE*, pages 273–282, 2011.