



FP6-IST-507219

PROSYD

Property-Based System Design

Instrument: Specific Targeted Research Project

Thematic Priority: Information Society Technologies

Evaluation of tools developed for error localization

(Deliverable 2.3/2)

Due date of deliverable: December 31, 2006

Actual Delivery date: December 31, 2006

Start date of project: January 1, 2004

Duration: 3 years

Organisation name of lead contractor for this deliverable: IBM

Revision: 1.0

Project co-funded by the European Commission within the Sixth Framework Programme (2000-2006)		
Dissemination Level		
PU	Public	<input checked="" type="checkbox"/>
PP	Restricted to other programme participants (including the Commission Services)	<input type="checkbox"/>
RE	Restricted to a group specified by the consortium (including the Commission Services)	<input type="checkbox"/>
CO	Confidential, only for members of the consortium (including the Commission Services)	<input type="checkbox"/>

Notices

For information, contact shirim@il.ibm.com

This document is intended to fulfil the obligations of the PROSYD project concerning deliverable, described in contract number 507219.

The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

© Copyright PROSYD 2004-2006. All rights reserved.

Table of Revisions

Version	Date	Description and Reason	By	Affected Sections
0.1	26.11.2006	Creation and integration of contribution from partners	Shiri Moran, Stefan Staber, Klaus Winkelmann, Karen Yorav	All
0.2	10.12.2006	Review	Karen Yorav	All
0.3	25.12.2006	Draft	Karen Yorav	All
1.0	26.12.2006	final version	Karen Yorav	Formatting

Authors

Klaus Winkelmann

Stefan Staber

Shiri Moran

Karen Yorav

Executive Summary

Property verification tools yield a trace as evidence that a design violates a property. Getting from the trace to the cause of the failure can be very time consuming. The property-based error localization tool, **BuFi**, is aimed at automating the localization of a fault. This work describes case studies performed on this tool by both IBM and OneSpin. The contribution and usefulness of the tool are evaluated; and the conditions under which the tool is beneficial are discussed.

Purpose

The purpose of this document is to describe the work done in case studies on the property-based fault localization tool BuFi. The document summarizes the experiences of using the tool on our case studies, and draws conclusions.

Intended Audience

This document is intended for researchers working on fault localization tools, as well as for formal verification engineers and hardware designers who use such tools. Basic knowledge the property specification language PSL is assumed.

Background

In deliverable 2.2/2, a novel **error localization** approach for helping a designer detect bugs, based on a failure of a property, is presented. A tool, called *BuFi*, that implements this approach was constructed by TU Graz, as part of PROSYD task 2.2. This document presents case studies of this tool.

Contents

Table of Revisions	iii
Authors.....	iii
Executive Summary	iii
Purpose.....	iii
Intended Audience	iii
Background.....	iii
Table of Figures	v
List of Tables	vi
Glossary	vii
1 Introduction.....	1
2 The Bufi Tool.....	2
Tool Description	2
Usage and Features	3
3 OneSpin Case Studies.....	4
Overview.....	4
Case Study “Arbiter”	4
Case Study “Protocol Processor”	7
4 IBM Case Studies	8
The Buffer example	8
Design description	8
Results for BUFFER.....	10
The FIFO example.....	15
5 Conclusions.....	16
6 References.....	18

Table of Figures

Figure 1: The BUFFER block diagram.....	9
Figure 2: The injected fault in BUFFER	9
Figure 3: A snapshot of a RuleBasePE trace of the error in BUFFER.....	9

List of Tables

Table 1: Results for Test 1, using assertions 1-5	11
Table 2: Results for each of the failing properties, with assertions 1-4.....	12
Table 3: Results for all failing properties, with assertions 1-8	13
Table 4: Results when using assertions 1-5 and 9-10.....	14
Table 5: Results with all assertions.....	15

Glossary

Assertion

A property that is expected to hold true on a specific design.

Behaviour

A succession of states of a design.

BuFi

A tool for automatic localization of fault candidates for sequential circuits at the gate or HDL level. The tool was developed by Stefan Staber of TU Graz.

Design

A model of a piece of hardware, described in some hardware description language (HDL).

Failure

An incorrect result. For example, a computed result of 12 when the correct result is 10.

Fault

An incorrect step, process or data definition. For example, an incorrect instruction in a computer program.

HDL (Hardware Description Language)

One of several specialized high-level languages used by semiconductor designers to describe the features and functionality of chips and systems prior to handoff to the IC layout process. Currently, the two standard HDLs in use worldwide are Verilog HDL and VHDL.

Property

A collection of logical and temporal relationships between expressions involving design signals, that represents a set of behaviours.

PSL

Property Specification Language, the language for specifications of designs upon which PROSYD is based.

SAT

Boolean satisfiability problem.

Sequential Circuit

A logic circuit whose output is a function of the present input and the previous state of the circuit.

Specification

The process of defining the expected behaviour of a hardware design.

Verification

The process of falsifying or verifying the functional and performance requirements of a design, be it chip, board or system. Many different kinds of verification tools are in use today, including simulation, formal verification, emulation and rapid prototyping.

1 Introduction

Verification tools usually provide a trace if the design does not fulfil a property. But even with this trace, it can be hard to find the fault contained in the system. In PROSYD deliverable 2.2/2 [2], a novel **error localization** approach for helping a designer detect bugs (given a trace as above) is presented. A tool, *BuFi* (BUG FInder), that implements this approach was constructed by Stefan Staber of TU Graz.

This document presents case studies, performed by both IBM and OneSpin, to evaluate the tool. The main method to evaluate the tool is to run it on a buggy version of a (small) design. The case studies also include a theoretical analysis of a larger design, which is instrumental in evaluating the fault model used by BuFi. These case studies demonstrate cases in which the tool could be beneficial as well as some limitations of the tool.

Loosely speaking, the case studies performed by OneSpin focus on analyzing the cases in which the BuFi tool is applicable. In fact, as a result of the OneSpin case study the applicability of the tool was enhanced. The case studies of IBM analyze how to use the tool efficiently.

The case studies suggest that error localization works when the fault is a result of a mistake in a single line, and especially when the tool is provided with a full set of specifications. Currently the tool is not applicable when bugs arise from a combination of errors in several lines of code, and it obviously cannot pinpoint conceptual mistakes. In the conclusions Section, we present our insights on how to use the tool efficiently, and to which directions we would like to see the research continue.

2 The Bufi Tool

Tool Description

Bufi (Bug Finder) is a tool for automatic localization of fault candidates for sequential circuits at the gate or HDL level. Bufi is a command line tool implemented on top of the model checker Vis. It takes as input a faulty design, a specification (a set of safety properties), and a set of counterexamples and returns a set of possible fault locations.

Bufi builds on the theory of Model based diagnosis, where a failure is seen as a discrepancy between the required and the actual behaviour of the system. The diagnosis problem is to determine those components that explain the discrepancy when assumed that they are incorrect. Such a component is called a diagnosis.

In the Bufi tool we state the diagnosis problem as a satisfiability (SAT) problem. In our setting, a counterexample of length k is given that proves that a specification does not hold. As in bounded model checking, we unroll the circuit to length k and build a propositional formula to decide whether the PSL specification holds. If we fix the inputs in the unrolled circuit to the values given in the counterexample and assert that the property holds, we arrive at a contradiction. The problem of diagnosis is now the problem of resolving this contradiction.

To this end, we extend the model of the circuit. We introduce a set of predicates, called abnormal predicates, which assert that a component functions incorrectly. Components are expressions at the HDL level. If an abnormal predicate is asserted, the functional constraints between inputs and outputs of the component are suspended. The diagnosis problem corresponds to determining which abnormal predicates need to be asserted in order to resolve the contradiction.

The fault model employed corresponds to the expression level. For example, a fault might be the right hand side of an assignment, the condition of an if-statement, or the conditions in a case statement. If the defect in the code is part of the fault model the tool guarantees that the faulty line is one of the provided fault candidates.

Bufi delivers single fault diagnoses. This means that the tool looks for single components that explain the discrepancy between faulty behaviour and the given specification. A single fault diagnosis might not exist, if there are multiple faults in the given design.

Usage and Features

Bufi is invoked from the command shell and takes as input a Verilog design, a specification, and a finite counterexample that violates a property in the specification. The output of the tool is a set of fault candidates in the design. The diagnosis engine of Bufi is based on the model checker Vis, which takes as input a design written in Verilog or in Blif-MV. Bufi handles the same input set as Vis. The specification contains a set of safety properties written in the linear-part of PSL. The counterexample file holds a finite trace written in Vis style. The output of Bufi contains the total number of diagnoses, and lists for each diagnosis the line of the original source and the corresponding line number.

3 OneSpin Case Studies

Overview

This chapter documents OneSpin’s efforts in evaluating the error localization method, algorithm, and tool. The work was carried out in cooperation with TU Graz. OneSpin analyzed the debugging process and its problems on an industrial verification project and provided representative examples on which TU Graz tested the algorithms.

The original intention of using the Protocol Processor (by Infineon) as the only working example was modified as follows. The protocol processor was considered too complex for a first analysis of the power of the error localization algorithm. Therefore a simpler design was selected, namely a priority-driven arbiter as it is used in many designs for sharing a resource between a (configurable) number of masters. The particular implementation used is owned by OneSpin and used for beginner-level training in property checking.

The first experiment described below led TU Graz to extend and generalize the algorithm, and resulted in correctly localizing errors for the arbiter. It demonstrated the capabilities, but also some limitations of the approach. As these limitations made it unlikely that new insights were to be gained from a larger case study, and as most of the planned effort was spent on this first case study, the case study “protocol processor” was then limited to a theoretical consideration rather than a practical application of the algorithm.

Case Study “Arbiter”

To test the algorithm, a buggy version of an arbiter was designed by deleting one line from a correct implementation – a line which resets the grant vector before entering the arbitration cycle. This is a realistic simulation of “forgetting” an assignment.

The VHDL version is as follows.

```
-- Arbiter design - OneSpin training
library ieee;
use ieee.std_logic_1164.all;

entity arbiter is
  generic (
    nr_masters: natural := 3
  );
  port (
    reset    : in  std_ulogic;
    clk      : in  std_ulogic;
```

```

    request_i: in  std_ulogic_vector(nr_masters-1 downto 0);
    free_i   : in  std_ulogic;
    grant_o  : out std_ulogic_vector(nr_masters-1 downto 0)
  );
end arbiter;

architecture rtl_min of arbiter is
  type state_t is (IDLE, START, BUSY);
  constant zero_c: std_ulogic_vector(nr_masters-1 downto 0) :=
    (others => '0');
  signal prio_s : std_ulogic_vector(nr_masters-1 downto 0);
  signal grant_s : std_ulogic_vector(nr_masters-1 downto 0);
  signal state_s: state_t;
begin
  nextstate_output: process(clk, reset)
    variable index_v: natural range 0 to nr_masters-1;
  begin
    prio_s <= (others => '0');
    -- compute the priority vector
    for index_v in 0 to nr_masters-1 loop
      if request_i(index_v) = '1' then
        prio_s(index_v) <= '1';
        exit;
      end if;
    end loop;
    if reset = '1' then
      state_s <= IDLE;
      grant_s <= (others => '0');
    elsif clk'event and clk = '1' then
      -- state machine
      -- BUG:
      -- the statement "grant_s <= (others => '0');" is missing here

      case state_s is
        when IDLE =>
          if prio_s /= zero_c then
            state_s <= START;
            grant_s <= prio_s;
          end if;
        when START =>
          state_s <= BUSY;
        when BUSY =>
          if free_i = '1' then
            state_s <= IDLE;
          end if;
        end case;
      end if;
    end process nextstate_output;
    grant_o <= grant_s;
  end rtl_min;

```

OneSpin provided this code to TU Graz, who had to modify it slightly to cope with the syntactic capabilities of the tool. In addition, a failing property was provided – stating that there is no grant without a request, which is obviously violated by this buggy design. Indeed the error localization algorithm came up with some diagnoses – but the correct one was not among them. This was no surprise as the error model used in this initial version of the algorithm did not take “missing-line“ errors into account – its fault model was limited to incorrect expressions. That is, it was examined whether an existing expression in the program could be changed so that the property holds.

Further, it turned out that the failure of a single property provides too little information for a focused diagnosis, and a more complete set of properties is needed.

Consequently, TU Graz developed two improvements

1. Extend the fault model to handle forgotten assignments,
2. Strengthen the property to be more realistic, i.e. describe the required behaviour.

The new version of the algorithm came up with a very interesting result: all its diagnoses were correct, i.e. each provided a correction that fixed the bug and maintained the correct functionality – indeed there are several ways to fix the bug, some of which were not initially obvious. These are inserted in **bold** in the following copy of the RTL code. In fact “diagnosis 5” was the expected one, and the others show other correct bug fixes (each diagnosis is shown as a bug fix consisting of one source code line to be inserted).

```
architecture rtl_min of arbiter is
  type state_t is (IDLE, START, BUSY);
  constant zero_c: std_ulogic_vector(nr_masters-1 downto 0) :=
    (others => '0');
  signal prio_s : std_ulogic_vector(nr_masters-1 downto 0);
  signal grant_s : std_ulogic_vector(nr_masters-1 downto 0);
  signal state_s: state_t;
begin
  nextstate_output: process(clk, reset)
    variable index_v: natural range 0 to nr_masters-1;
  begin
    -- diagnose 1: grant_s <= (others => '0');
    prio_s <= (others => '0');
    -- compute the priority vector
    -- diagnose 2: grant_s <= (others => '0');
    for index_v in 0 to nr_masters-1 loop
      -- diagnose 3: grant_s <= (others => '0');
      if request_i(index_v) = '1' then
        prio_s(index_v) <= '1';
        exit;
      end if;
    end loop;
    -- diagnose 4: grant_s <= (others => '0');
    if reset = '1' then
      state_s <= IDLE;
      grant_s <= (others => '0');
    elsif clk'event and clk = '1' then
      -- state machine
      -- diagnose 5: grant_s <= (others => '0');
      case state_s is
        when IDLE =>
          if prio_s /= zero_c then
            state_s <= START;
            grant_s <= prio_s;
          end if;
        when START =>
          state_s <= BUSY;
        when BUSY =>
          if free_i = '1' then
            state_s <= IDLE;
          end if;
        end case;
      end if;
    end process nextstate_output;
    grant_o <= grant_s;
  end rtl_min;
```

Open questions arising from this experience are:

- will it always or at least usually be guaranteed that an obvious (to a human) solution will be proposed when one exists?
- if not, then what does this mean for the maintainability of the source code down the line, as it fills up with non-obvious fixes?

Answering this will require further and more realistic examples to be addressed.

Case Study “Protocol Processor”

This industrial IP design was verified completely using OneSpin's technology, and several serious bugs were discovered. OneSpin did an analysis of the bugs found with respect to the required fault model. Candidate fault models considered were

- wrong expression (to be assigned a different expression),
- missing statement (to be inserted),
- extra statement (to be deleted).

Unfortunately none of the serious bugs fell into any of these categories. Indeed the difficulty and complexity of a processor design makes it unlikely that a bug can be fixed by a simple one-line modification – except for trivial omissions, bugs are made rather at a conceptual level and require several source locations to be modified.

On the other hand OneSpin analyzed the effort invested in verification and diagnosis, with the following results: Whenever a property failed, some effort is involved in confirming the bug, i.e., ensuring that the property itself is correct, and that the design indeed violates the specification. This effort ranges between 2 hours to 2 days. However, after this confirmation it is in most cases not difficult to localize the bug, i.e., to understand its cause at the RT level and fix it. This effort ranges between half a day to one day.

Thus, while a good automated error localization tool would be nice to have, it is not considered among the most pressing issues in the overall verification activity.

4 IBM Case Studies

This chapter documents IBM's efforts in evaluating the error localization tool. In general, the idea is to select a suitable design, intentionally modify it to inject a fault, and then use BuFi to localize the fault. We made sure to stay within the limitations of the tool in terms of its fault model. We present our conclusions and recommendations regarding the tool, the methodology, and the fault model.

The Buffer example

Design description

IBM selected a simple BUFFER design that was intentionally modified to be erroneous by changing a single line of code¹. This design is a tutorial used by IBM for beginner-level RuleBasePE training. In cooperation with TU Graz, the Verilog code of the design was modified to fit the Verilog format accepted by Vis. The error localization tool was then used in several ways to evaluate its performance, and to develop a methodology for using it effectively.

BUFFER is a design block that buffers a word of data (32 bits) sent by a sender to a receiver. It has two control inputs, two control outputs, and a data bus on each side, as shown in Figure 1. Communication (on both sides) takes place by means of a four-phase handshaking protocol as follows:

- When the sender has data to send to the receiver, it initiates a transfer by putting the data on the data bus and asserting **STOB_REQ** (sender to buffer request). If BUFFER is free, it reads the data and asserts **BTOS_ACK** (buffer to sender acknowledge). Otherwise, the sender waits. After seeing **BTOS_ACK**, the sender may release the data bus and deassert **STOB_REQ**. To conclude the transaction, BUFFER deasserts **BTOS_ACK**.
- When BUFFER has data, it initiates a transfer to the receiver by putting the data on the data bus and asserting **BTOR_REQ** (buffer to receiver request). If the

¹ IBM's intention was to use the GenBuf example, which is much larger and more complicated. However, it turned out that Vis cannot handle this example. To get it to run with Vis, IBM would have to manually re-code the design from scratch in the language that Vis supports, which is beyond the scope of this case study.

receiver is ready, it reads the data and asserts **RTOB_ACK** (receiver to buffer acknowledge). Otherwise, BUFFER waits. After seeing **RTOB_ACK**, BUFFER may release the data bus and deassert **BTOR_REQ**. To conclude the transaction, the receiver deasserts **RTOB_ACK**.

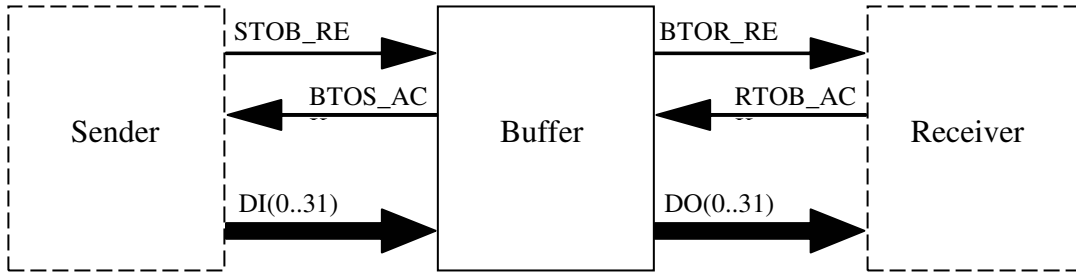


Figure 1: The BUFFER block diagram

The BUFFER design was intentionally modified to be erroneous, by changing a single line of code, shown in Figure 2.

Correct line (#50)	Erroneous line (#50)
if (!RTOB_ACK && occupied)	if (occupied)

Figure 2: The injected fault in BUFFER

As a result of this modification, BUFFER can initiate a new transaction with the receiver while a transaction with the receiver is still going on; more specifically, BUFFER can erroneously request a new transaction with the receiver (by asserting **BTOR_REQ**) while the receiver's acknowledgment (**RTOB_ACK**) of the previous transaction is still asserted. This acknowledgment is mistakenly interpreted as an acknowledgment for the new transaction. Of course, this introduces illegal behaviour, as demonstrated by the trace in Figure 3.

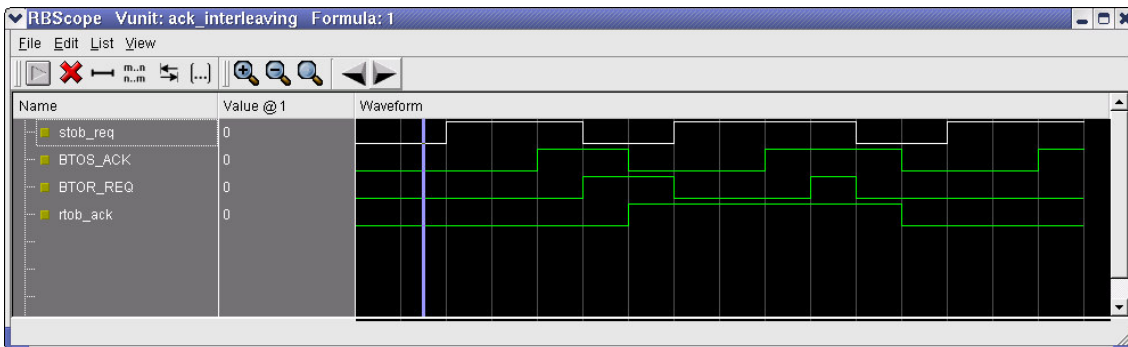


Figure 3: A snapshot of a RuleBasePE trace of the error in BUFFER

The BUFFER code has 33 effective lines of code (possible places for faults). The implementation of the design has 36 flip-flops, 317 gates, and 37 inputs.

Results for BUFFER

To evaluate the error localization tool we performed five different tests, experimenting with different inputs to the tool. We started with a list of five properties that are meant to hold on the design. The above fault injection causes three of the properties to fail. All five tests involve localizing the fault for each of the three failing properties separately, and using the same counterexamples. The difference between the tests is the set of properties that is given to Bufe for localization. As explained in Section 2, Bufe's input consists of a counterexample and a set of assertions. Each diagnosis that the tool outputs is an expression that, if changed, can cause the whole set of assertions to pass. Theoretically, it is expected that adding more assertions will improve Bufe's results, i.e., reduce the number of diagnoses. Our experiments show this, while developing a methodology for working with the tool. The sum of all our experiments boils down to a set of recommendations that can be used by future designers working with error localization.

Test 1

The first test considered the following five PSL assertions. These are linear time properties describing the correct behaviour of the design².

- (1) "RTOB_ACK is asserted between any two BTOS_ACK assertions"
always (!RST & rose(BTOS_ACK) ->
next (rose(RTOB_ACK) before rose(BTOS_ACK))
- (2) "BTOS_ACK is asserted between any two RTOB_ACK assertions"
always (!RST & rose(RTOB_ACK) ->
next (rose(BTOS_ACK) before rose(RTOB_ACK))
- (3) G (!RST & rose(BTOS_ACK) & DI(0)=0 ->
next_event (rose(RTOB_ACK)) (DO(0)=0));
- (4) G (!RST & rose(BTOS_ACK) & DI(0)=1 ->
next_event (rose(RTOB_ACK)) (DO(0)=1));
- (5) "BTOR_REQ is active weak_until RTOB_ack"
always (!RST & BTOR_REQ -> BTOR_REQ until RTOB_ACK);

The first two assertions claim that neither overflow (two reads without a write in between) nor underflow (two writes without a read in between) can occur. While the error in the design does not affect the second assertion, it clearly affects the first one. Indeed, assertion 1 fails and assertion 2 passes. The trace of Figure 3 demonstrates a failure of assertion 1.

The third and fourth assertions verify data integrity, i.e., that the data sent to the receiver is the same data received from the sender. To simplify the verification

² Note that this is not a complete set in the sense that it does not describe all aspects of the design.

process we test a single bit instead of the whole vector. Our injected fault causes both of these assertions to fail.

The fifth assertion states that “buffer to receiver request” (**BTOR_REQ**) stays stable until the corresponding “receiver to buffer acknowledgment” (**RTOB_ACK**) is granted. Its validity is not affected by the injected fault.

We used the Bufe tool to evaluate each of the failed assertions, while requesting that all 4 assertions hold after the bug fix. In each run we generate a counterexample to a single assertion, and then run Bufe to localize the fault with all assertions together. The results are presented in Table 1.

Assertion #	Counterexample length	Number of diagnoses	percentage of the code
1	14	10	30%
2	PASS	-	-
3	16	10	30%
4	16	10	30%
5	PASS	-	-

Table 1: Results for Test 1, using assertions 1-5

A detailed analysis of the results reveals the following:

- The three sets of diagnosis are the same.
- The set of diagnoses of all of the assertions included the erroneous line.
- When examining the diagnoses that do not point to the fault that we injected, we can see that it is possible to modify each of these lines so that all four properties hold. This conforms to what the tool guarantees. Obviously, such changes would introduce new bugs. This is because the set of assertions is not complete – the conjunction of these five assertions allows for behaviours that are incorrect for the buffer.

This naive test produced a list of diagnoses that points to 30% of the code. In general, this number is too high. However, in the Buffer example this is not a lot of code. It is difficult to extrapolate from this to larger designs. Instead, we wish to examine whether the user can influence the results, and with what effort.

Test 2

It is obvious that Bufe’s results are influenced by the set of properties that are given as input. In this test we examine the extent of this effect. We re-ran the first test without assertion 5. This means that a fault candidate is now required to make assertions 1 – 4

pass, but is not required to make assertion 5 pass. Table 2 presents the results for each of the failing properties.

Assertion #	Counterexample length	Number of diagnoses	percentage of the code
1	14	15	45%
3	16	12	36%
4	16	12	36%

Table 2: Results for each of the failing properties, with assertions 1-4

It is easy to see that the number of diagnoses for each failing property is considerably increased, most notably for assertion 1. From this we draw the conclusion that it is worthwhile writing more specifications in order to receive higher quality results from error localization. This adds yet another incentive to the long list of reasons why specification is an important aspect of the design flow.

In addition, a more detailed analysis of the results reveals that:

- The set of diagnoses of the third and fourth assertions are the same. This makes sense, since these two assertions state the same property with different data values.
- The set of diagnoses of assertion 1 is a superset of the set of diagnoses of assertion 3 (and assertion 4). This also makes sense, since any violation of assertion 1 implies a violation of assertions 3 and 4.

As a matter of fact, these two observations hold for all of the tests that we performed on the BUFFER example.

Test 3

This test is based on ideas that we developed while working with the tool, in an attempt to produce more quality fault candidates. We noticed that in many cases, an expression becomes a fault candidate because, when changed, it disables legal behaviours of the design. For example, assume we have a property

“always (event -> something_good)”

Assume also that “event” is generated only in the “then” part of some “if” condition:

```

if (exp) then
    generate(event)
else
    ...

```

In this case “exp” is a fault candidate because making it “false” causes the property to pass. Obviously, this is not what we are looking for. We want candidates that allow

for interesting behaviours that are correct. This observation led us to suggest using vacuity.

Loosely speaking, a property passes *vacuously* if it does so regardless of the satisfaction of one of its sub-formulas. The precise definition can be found in [1]. For example, the example property above passes vacuously when “event” never happens, because in this case the right hand side of the implication is never checked.

In general, non-vacuity is asserted using liveness properties. Non-vacuity of assertions 1-5 from test 2 would be asserted by the following three properties:

```
"BTOS_ACK is eventually asserted"
  eventually! ( !RST & rose(BTOS_ACK))

"RTOB_ACK is eventually asserted"
  eventually! ( !RST & rose(RTOB_ACK))

"BTOR_REQ is eventually asserted"
  eventually! ( !RST & BTOR_REQ)
```

BuFi operates on a bounded depth, in which case non-vacuity can be expressed using *bounded-liveness* properties. These are actually safety properties. We supplied BuFi with the following three properties, which are bounded-liveness versions of the non-vacuity assertions:

```
(6) "BTOS_ACK is eventually asserted"
     F[k] ( !RST & rose(BTOS_ACK))

(7) "RTOB_ACK is eventually asserted"
     F[k]( !RST & rose(RTOB_ACK))

(8) "BTOR_REQ is eventually asserted"
     F[k] (!RST & BTOR_REQ)
```

where $F[k]$ is shorthand for “holds in at least one of the next k clock cycles”. So:

$$F[3] p \equiv p \vee (X p) \vee (XX p) \vee (XXX p)$$

Our counterexamples are all of length less than or equal to 16, so we used $k=16$.

As before, we ran BuFi for each of the failed assertions (1, 3, and 4), while requesting that all 8 assertions hold after the bug fix. The results are presented in Table 3.

Assertion #	Counterexample length	Number of diagnoses	Percentage of the code
1	14	10	30%
3	16	9	27%
4	16	9	27%

Table 3: Results for all failing properties, with assertions 1-8

We notice that Test 3 gives better results than Test 1 for assertions 3 and 4, while giving the same results for assertion 1. These results lead us to believe that vacuity could be a useful tool for improving the quality of fault candidates. Although we applied manually, this idea can easily be automated and implemented within the tool.

Test 4

This test is a second attempt to improve the quality of the diagnoses list compared with test 1. This time we added assertions that limit the behaviours of the design when combined with its (non-deterministic) environment.

In general, the design is verified with a non-deterministic environment that is capable of driving all possible legal input traces. In tests 1 and 2 the fault localization tool was supplied with a set of assertions that are expected to hold in all executions. In the current test we attempt to make the localization algorithm more efficient by examining a sub-space of all possible executions. The rationale is that when faced with a bug the designer can easily produce assertions that describe a simplification of the general behaviour, and check whether the bug still exists. Looking at a simple sub-space is a common method for debugging. In the case of error localization, this is achieved by adding assertions that hold only on a subset of the legal inputs supplied by the environment.

In this test we used assertions 1-5 from test 1, plus the following two assertions:

- (9) "RTOB_ACK is high only if BTOR_REQ was high the previous cycle"
always (next(RTOB_ACK=1) -> (BTOR_REQ=1));
- (10) "BTOS_ACK is high only if STOB_REQ was high the previous cycle"
always (next(BTOS_ACK=1) -> (STOB_REQ=1));

Note that in this test we do *not* use assertions 6-8.

Assertions 9 and 10 state that both acknowledgments (**BTOS_ACK** and **RTOB_ACK**) should be deasserted no later than one clock cycle after the corresponding request (**STOB_REQ** and **BTOR_REQ**, respectively) is deasserted. These assertions are not implied by the description of the communication protocol of BUFFER. Assertion 9 fails on our example because the buffer's environment allows the general case. The BUFFER itself is a specific implementation that satisfies assertion 10.

Again, we used the tool to evaluate each of the failed assertions (1, 3, and 4), while requesting that all seven assertions hold after the bug fix. The results are presented in Table 4. Indeed we find that limiting the behaviour of the environment improved the quality of the error localization results.

Assertion #	Counterexample length	Number of diagnoses	Percentage of the code
1	14	9	27%
3	16	9	27%
4	16	9	27%

Table 4: Results when using assertions 1-5 and 9-10.

Test 5

In this test we provided the tool with all the assertions 1 – 10. The results are presented in Table 5.

Assertion #	Counterexample length	Number of diagnoses	Percentage of the code
1	14	9	27%
3	16	8	24%
4	16	8	24%

Table 5: Results with all assertions

This turned out as the best execution. It shows once again that adding assertions significantly improves the accuracy of the results.

The FIFO example

This example is a real design that we received from Marvell [3]. We use it often as a case study because the source is open and we can freely distribute it outside of IBM. It was especially suitable for this case study because it is written in Verilog.

The FIFO was designed to bridge between two clock domains. In the original setting it was meant to pass data from one clock domain to another, under the assumption that the clock frequencies are very close. As long as the difference between the two clock frequencies is not too large the FIFO works perfectly; when the frequency of the sending side is too fast the FIFO may lose information.

The first obstacle we encountered was that Vis does not support multiple clock domains. This was easily solved by connecting both clocks to a single clock. The result is a design that is a simple FIFO and functions correctly. The next step was to inject a fault into the design. We were limited to single component faults. A suitable “if” expression was chosen so that when the condition of this “if” is negated the FIFO displays an erroneous behaviour on a certain scenario, but most of its properties remain in tact. This fault mimics a typo, or a misconception, that is not easily corrected because it does not display itself often.

The example, as it is, does not conform to the limited Verilog subset that is supported by Vis. In cooperation with TU Graz we attempted to change it so that Bufi could be used. There were several technical difficulties to do this, mostly because it is a real design that uses complicated Verilog constructs for reasons of performance. It turned out that to get this example to work we would be required to re-write it completely, which was not possible with our given resources.

5 Conclusions

The fault localization tool is a first prototype of results for a very new research area. Fault localization and correction is an ambitious goal that has not been pursued for very long. As such, we evaluated this tool as a first stepping stone rather than a complete tool. We focus more on its potential than on its current capabilities. In this respect, we differentiate between capacity, and methodology. Capacity issues relate to the tools applicability and ease of use. How large a design can it handle? How easy is it for an engineer to use? What input languages does it accept? On the other hand, methodology issues relate to the tools contribution to the design cycle. How well can it localize a fault? Will it produce too many false negatives? Can it save time, or will it be a burden on the designer? Keeping in mind that we are experimenting with a prototype, we attempt to ignore capacity issues and focus on methodology issues.

Technicalities

To start with the least important, we note that both IBM and OneSpin found the tool difficult to experiment with. Due to the tool's reliance on Vis, actual designs could not be used. The manual work involved in re-writing a design so that it complied with the limited language Vis supports, forced both companies to use very small designs. Thus, our conclusions regarding the capacity of the tools must be regarded as preliminary. On the other hand, even while experimenting on a small scale we were able to draw interesting conclusions regarding the methodology for using the tool.

The number of diagnoses

The first and foremost issue that needs to be addressed is the number of diagnoses the tool produces. This is the most obvious measure of the tool's success. An automatic tool that can produce a small number of potential faults, while guaranteeing that one of these suggestions is the right one, would be extremely useful.

Here, we assume only faults that are covered by the fault model used by Bufe. In IBM's BUFFER design the injected fault was a change in a single expression, which is covered by Bufe's fault model. And indeed, in all variations Bufe succeeded in pointing at the faulty line. For this example the best results that we achieved were fault candidates that cover 24% of the code. This is obviously too much and cannot be expected to aid in debugging of a real design. However, this number is determined by the combination of the fault model and the design. A candidate fault is defined to be a single expression that can be changed so that all properties pass for a given bound. Bufe finds the exact set of all expressions in the design that adhere to this definition.

A re-examination of the BUFFER example reveals that it is so simplistic that indeed almost any line can be changed in some way so that the properties pass. We believe this situation is not likely to occur on larger, more realistic, designs. Still we remark that, in our opinion, in order to be helpful we would expect the tool to point at a small number of locations, which should be the subject of further research.

Based on our experimentation, we suggest a few ways in which the number of diagnoses can be reduced:

- A complete set of specifications is extremely useful to have. This is good methodology in general, but does not happen in practice. In the case of error localization, it is important to have as large a set of properties as possible, and ensure that the properties cover all aspects of the design's functionality.
- When given too many diagnoses, one can improve the results by adding stronger assertions. We found that adding assertions that limit the environment to a common case, one that still allows for the erroneous behaviour, can improve the precision of the localization algorithm.
- In many cases, an expression becomes a fault candidate because, when changed, it disables many legal behaviours of the design. Consequently, the properties that check these behaviours pass vacuously. We suggest adding to the property set a supplemental set of properties that imply non-vacuity. Indeed, we found that adding such assertions improved the precision of the algorithm. In the general case vacuity is checked using liveness properties. But since Bufe operates on a bounded depth, non-vacuity can be expressed using bounded-liveness properties.

The fault model

The most interesting conclusions from the case study are regarding the fault model. IBM's BUFFER example shows that the currently used fault model of single-line-errors can produce too many fault candidates. On the other hand, OneSpin's analysis of real faults clearly shows that this is not the fault model that is needed.

Error localization can be a useful addition to the design flow under three conditions:

1. A formal specification is available describing all relevant functionality
2. The type of the bug is anticipated in a fault model.
3. The fault localization algorithm handles the complexity arising from the space of admissible diagnosis.

While condition 1 can be achieved by a good property-based specification, conditions 2 and 3 are more challenging in practice: first they are competing with each other – the more powerful the fault model, the more complexity arises. Further it is not clear how a realistic fault model covering a substantial share of practical bugs should look like. Finally, performance data for such realistic scenarios are not yet available.

Following OneSpin's analysis we find that the fault model needs to include missing statements and missing branching commands.

6 References

- [1] I. Beer and S. Ben-David and C. Eisner and Y. Rodeh.
“Efficient Detection of Vacuity in Temporal Model Checking”, Formal
Methods in System Design 18(2), 2001.
- [2] R. Bloem and S. Staber.
“Property-based error localization”, April 2005. PROSYD D2.2/2.
- [3] Marvell Technology Group Ltd. URL: <http://www.marvell.com>.