FudgeFactor: Syntax-Guided Synthesis for Accurate RTL Error Localization and Correction

Andrew Becker[1(✉)], Djordje Maksimovic[2], David Novo[1], Mohsen Ewaida[1], Andreas Veneris[2], Barbara Jobstmann[1], and Paolo Ienne[1]

[1] Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
andrew.becker@epfl.ch
[2] University of Toronto, Toronto, Canada

Abstract. Functional verification occupies a significant amount of the digital circuit design cycle. In this paper, we present a novel approach to improve circuit debugging which not only localizes errors with high confidence, but can also provide semantically-meaningful source code corrections. Our method, which we call FudgeFactor, starts with a buggy design, at least one failing and several correct test vectors, and a list of suspect bug locations. We obtain the suspect location from a state-of-the-art debugging tool that includes a significant number of false positives. Using this list and a library of rules empirically characterizing typical source-code mistakes, we instrument the buggy design to allow each potential error location to either be left unchanged, or replaced with a set of possible corrections. FudgeFactor then combines the instrumented design with the test vectors and solves a 2QBF-SAT problem to find the minimum number of source-level changes from the original code which correct the bug. Our 13 benchmarks demonstrate that our method is able to correct a sizable portion of realistic bugs within a reasonable computational time. With the aid of available golden reference designs, we show that those corrections are, at least on these benchmarks, always valid and non-trivial fixes. We believe that our technique significantly improves over other debugging tools in two respects: When we succeed, we obtain a much more precise bug localization with no false positives and little or no ambiguity. Additionally, we offer bug corrections that are inherently meaningful to the designers and enable designers to quickly recognize and understand the root cause of the bug with a high level of confidence.

1 Introduction

Functional verification is a traditionally thorny process which occupies up to two thirds of the digital circuit design cycle [9]. There are at least two ways to reduce the time spent on ensuring functional correctness: either ease the process of developing functionally-correct circuits from the beginning, or improve circuit debug and verification tools. This paper takes the latter approach. Although formal verification tools typically return a counterexample when verification fails,

N. Piterman (Ed.): HVC 2015, LNCS 9434, pp. 259–275, 2015.
DOI: 10.1007/978-3-319-26287-1_16

the subsequent debugging process (i.e., error localization and correction) is typically lengthy and heavily reliant on designers' expertise and experience. Tools exist to help in error localization and correction, but most work on the subject has either suggested repairs at the netlist level [6,7], or tried to map netlist repairs back to RTL source code (e.g., [10,17]), which is not always possible and can lead to incomprehensible repair suggestions. We see this as problematic, as designers rarely work directly with netlists: even if tools find errors and suggest appropriate corrections in the netlist, designers must still spend an inordinate amount of time finding the true root cause at the register transfer level to be able to implement a correction they understand and can vouch for. Thus, we think it essential to locate and correct errors directly at the register transfer level, where designers typically work.

In this work, we present FudgeFactor, a syntax-guided synthesis tool for source-level error localization and correction. It takes as input a buggy circuit design, at least one failing test vector, some correct test vectors, and a list of suspect error locations. This list may come from any state-of-the-art error localization tools. These tools are usually remarkably efficient and can handle very large designs but lack precision. This leads to tens—or more—of fairly vague false-positive suspect locations. In our case, we use a commercial verification tool based on the work of Smith et al. [15] to obtain the list of suspect locations. Using this list and a library of rules characterizing typical source code mistakes, we automatically instrument the buggy design to allow each potential bug to either be left unchanged, or replaced with a set of possible corrections. FudgeFactor then combines the instrumented design with the test vector(s) and solves a 2QBF-SAT problem to find the minimum number of source-level changes from the original code which correct the bug. Because all correction rules describe semantically meaningful transformations, changes FudgeFactor presents to the user are highly likely to address the root cause and remove the error. Definitely, not all design errors are typical, "standard" mistakes, and thus our approach can never be complete, regardless of the number of rules in our library. Yet, we provide a quick, high-confidence initial debug pass which virtually eliminates a lengthy root cause analysis for a significant number of frequently recurring design errors. We have tested our tool with 13 different benchmarks from 3 real-world designs available on OpenCores [12] and demonstrate here that FudgeFactor suggests valid corrections for a sizeable portion of the bugs within a reasonable computational time.

FudgeFactor significantly owes to the approach used by Singh et al. to give meaningful automatic feedback to students of a programming course using Python [14]—in fact, the ability to "teach" the designer in which respect the design fails is exactly what drives our efforts and distinguishes our goal. Yet, our approach in the context of digital design results in at least a couple of significant advantages: (1) Our source-level correction-rules are not at all problem-specific but empirically represent an extensible library of typical mistakes that may occur in any design, such as using a wrong compatible signal in an expression, invoking the incorrect Boolean operator, or instantiating a wrong constant.

(2) The broadness of our rules is key to be able to debug arbitrary circuits but, if applied indiscriminately, would naturally catastrophically restrict our scalability. This does not happen because, in this domain, we can leverage many tools, some even commercial, which return approximate error location information using totally different techniques that happen to be scalable to industrial size designs. Thus, we only very selectively apply our generous set of correction rules to the candidate locations and, as our experiments show, incur perfectly acceptable run times.

The rest of the paper is organized as follows: We discuss additional related work in Sect. 2. Sections 3 and 4 describe FudgeFactor in more detail: the former addresses the basic methodology while the latter describes how we solve some fundamental scalability issues. We present our experimental setup in Sect. 5 and discuss results in Sect. 6. Finally, Sect. 7 concludes the paper.

2 Related Work

Debugging of hardware designs has been studied extensively in the previous three decades. This field typically focuses on two related but distinct facets of the problem: finding potential error locations (at whatever level of the design), and proposing corrections which eliminate the errors.

Error Localization. Early works on design error localization were targeted at gate-level representations. Work by Chung et al. [6,7] proposed a localization technique which expresses the problem as a set of Boolean equations, where the existence of a solution determines if the gate or wire is a potential error source or not. Smith et al. [15] improved on the scalability and quality of gate-level error localization using Boolean satisfiability (SAT). Fixing design errors at the gate level produces obscure corrections that are very hard for the circuit designer to understand. Our approach tackles the problem of returning meaningful corrections for the designer. Given the popularity of HDLs among hardware designers, source-level error localization has become increasingly attractive. Works by Bloem et al. [3] and Peischl et al. [13] discussed *Model-Based Diagnosis (MBD)* methods for error localization in VHDL descriptions. Several works [5,15] adapted the concept of gate-level fault modeling to source-level error localization by mapping gates to their HDL description. Our approach adopts the same concept of inserting multiplexers, but instead of having a single free signal, we insert proper error corrections based on an error library model. In this way we restrict the number of possible solutions and improve solver scalability.

Error Correction. Error localization techniques usually generate a design component set: either RTL locations, gates in the netlist or combinational paths that can be modified to correct the error. Chang et al. [4] proposed an approach for correcting gate-level errors using signatures of candidate faulty gates. A signature is a list of bits each corresponding to the gate output for a given set of test vectors. Their approach corrects signatures and re-synthesizes them to replace the gate with one represented by the corrected signature. The idea

has been applied to source-level error correction and extended to hierarchal and sequential designs [5]. Jobstmann et al. [10] suggested an approach to correct erroneous Verilog designs. Like our work, this approach assumes access to a list of suspect error locations but uses a different error and reference model. It allows corrections that can be represented by arbitrary functions in terms of the state and input variables. This leads to a very general correction model at the expense of readability and reasonability of correction suggestions. We believe that our correction rules lead to corrections that are more meaningful and much easier to understand. In addition, their approach relies on a formal specification (given in Linear Temporal Logic) that describes the desired behavior of the design. Since formal specifications are often unavailable, we focus on simulation vectors, the de facto standard technique in industry to verify digital designs. Staber et al. [17] have extended the above-mentioned repair approach to error localization by assuming that only a location that can be corrected can be an actual error location. This approach is more precise but also more expensive than other error localization approaches. It is similar to our approach as it also aims to increase the precision of error locations by searching for correction suggestions. However, there are significant differences in the setup and underlying technique. Furthermore, our approach is a SAT-based technique, while their approach used BDD-based methodologies, which are known to be less scalable for large designs.

The related work probably most relevant to this paper are mutation-based approaches. *Mutations* were introduced by Debroy and Wong in the software world [8] and closely resemble our "fudging" rules (Sect. 3.1), but their mutations are extremely simple and success is determined with some test cases, with no formal verification. More recently, Alizadeh et al. [1] have used mutations to create potentially working hardware designs from a failing one; their mutations, essentially targeting signal processing designs, are a restricted and predetermined version of our rules, the latter being much more articulated and constituting an expansible library. And, again, successful mutations are identified by enumeration, whereas our use of 2QBF-SAT is more efficient and also corrects situations where multiple rules or mutations are needed for a single bug.

3 "Fudging" Buggy RTL Circuits

Figure 1 shows the complete FudgeFactor flow from a buggy RTL circuit to a (list of) suggested source-code correction(s) which fix the error(s) in the circuit. The buggy circuit must come with some test vectors and at least one of them must be failing and expose the error(s).

The approach behind FudgeFactor is *syntax-guided synthesis* [2]: we tweak (or "fudge") the original buggy RTL specification in many different ways to try to synthesize a new RTL specification which is syntactically as close as possible to the buggy one yet which does not exhibit the error, and is therefore a candidate correction. In the spirit of syntax-guided synthesis, we follow the intuition that acting at the source-code level, respecting the syntactic template provided by the human designer(s) who inadvertently introduced the error in the first place,

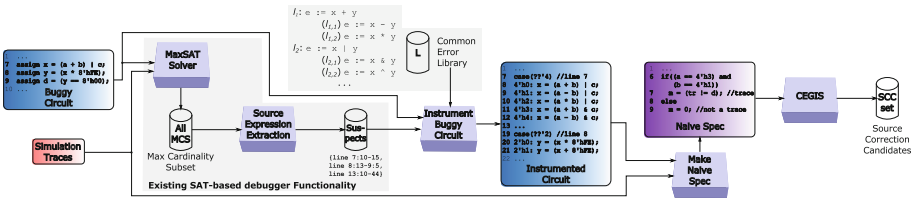

Fig. 1. The overall flow for FudgeFactor. The inputs are a buggy Verilog design and one or more error traces and the output are candidates to correct the RTL source code.

makes it possible to find good corrections much more easily. More specifically, in our application, we note how some erroneous RTL designs may be extremely "close" to the correct one in the syntactic space and yet fundamentally "far" in the netlist space. A case in point is a missing condition of assignment in a *case* construct: a one-character difference in RTL can have such a drastic effect as transforming a combinational circuit into an erroneous sequential circuit. As we will show, our approach is perfectly capable of providing *meaningful* corrections in such cases.

3.1 Common Error Library

The key intuition of our approach is that many of the errors we make as programmers and designers are relatively predictable in nature: we may mistake one signal for another one which is electrically compatible (i.e., the same number of bits and doesn't cause a logic loop), and this may happen both on the right side of an expression (a wrong input being used in the calculation) as well as on the left side (the wrong signal being assigned). Or we may compute a wrong logic or arithmetic function by replacing, for instance, an OR with an AND or specifying a subtraction for an addition. Or, as already mentioned, we may forget some clauses in a conditional statement, leading to a variety of errors at the netlist level including the potential for circuits (or subcircuits) to switch across the combinational to sequential border. In different contexts, researchers have already noted that this is an effective way to capture a large fraction of programming errors [14]. Self-evidently, this approach cannot capture all possible errors. For example, errors of omission (missing conditions in an expression, etc.) are unlikely to be corrected with our general rules. However, we think there is great practical value in efficiently capturing and correcting common errors and thus freeing precious designer time for concentrating on only a relatively few hard cases.

Our common error library has been developed by reflecting on our experience as RTL designers and by manually inspecting a large number of buggy designs, including student assignments and bug fixes in open-source RTL repositories. (We have excluded most of the circuits which we use as benchmarks; more details about this aspect are given in Sect. 5.) At this point, the extensible library contains only a few very general source-code trasformation rules

Table 1. The common error library rules currently implemented in FudgeFactor. Note that this is by no means a list of all rules one may add, or even an attempt at capturing all of the most common RTL errors. Also note that the transformer rules do not necessarily *replace* the subgraph matched on: the transformer rules insert the *possibility* of using such a change, for which it is often necessary to add multiplexers, signals, etc. to the AST.

Rule	Checker (if the subgraph looks like...)	Transformer (insert these options...)
A	Signal indexing operation	Indices and ranges may be shifted to the left or right by one
B	Incomplete case without default	Signals assigned in case get a default assignment of any compatible signal, or a pure free variable
C	If ... If ... Else assigning the same signal	Allow use of a parallel If ... Else If ... Else with the same conditions
D	Signal in any statement explicitly mentioned in candidate set	Allow referring instead to any compatible signal
E	A bitwise comparison operator	Allow comparing with any other bitwise comparison operator instead
F	A constant value on right-hand side; not an index/range	Allow using instead any constant value (a pure free variable)
G	A ternary expression	Allow using instead the same ternary expression, but with the condition inverted

described in Table 1. Although limited, it turns out that this is already very effective.

3.2 Error Modeling

Error rules model common designer errors as modifications to the *abstract syntax tree (AST)* obtained by parsing the input RTL. Because we work directly on the AST, our rules are not limited to identifying line-by-line modifications. Our rules can happily identify and propose corrections for errors spanning multiple lines. Each rule is composed of two different parts: The first part, the *rule checker*, determines whether the particular rule is applicable. The second part, the *rule transformer* expresses the modifications to the AST necessary to include a set of potential corrections. For example, the rule checker of the rule in Fig. 2 checks whether an AST node represents a bitwise OR operator. If the rule checker matches a particular node of the AST, the corresponding rule transformer is executed and the AST is modified. Figure 2 is a simplified example of a rule one might really want to implement; in practice, the rule checker would probably match all bivariate Boolean operator nodes and allow the choice of any shows

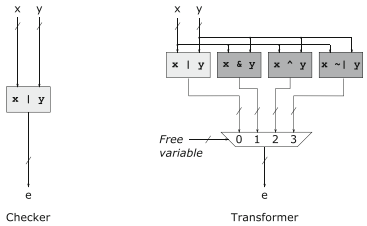

Fig. 2. A visual example of an error rule. The designer has written the expression $e := x \mid y$ and this rule suggests that what he or she *might have meant* was any other Boolean function (e.g., AND, XOR, NOR) instead of OR. The *rule checker* is represented in the left part of the figure and, in this elementary case, essentially says that this rule applies potentially to any OR operation. The right part of the figure is the *rule transformer*, which describes how the AST can be rewritten to allow the choice of such an alternative Boolean operator. Note that this figure shows, for convenience, the rule in the form of circuits, but rules are actually described and implemented using AST nodes and some rule-specific ad-hoc code.

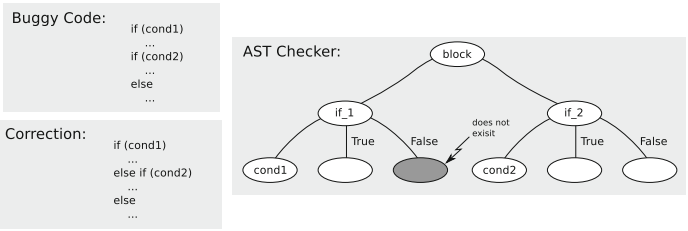

Fig. 3. A more complex rule checker. This rule checker is shown as an AST subtree to match in the design AST. It approximately corresponds to rule **C** expressing the fact that the designer might have forgotten an *else* clause in an *if* statement. This shows some of the advanced quantifiers we use in our rules, such as the fact that two *if* statement *must* exist in immediate succession within a block but the first one *must not* have already an *else* clause. The example is slightly simplified compared to the actual AST of the parser to improve readability.

the rule checker and rule transformer for a simplified version of such a rule matching only an OR operator.

Rule checkers can perform both structural and property checks. Structural checks are based on tree isomorphism (i.e., detecting if the structure of the AST subtree matches a reference one): they detect subtrees of interest and discard cases where the rule transformer should not be applied. We implement rule checkers programmatically, although we think that it could be possible (but not necessarily truly advantageous) to define a formal language syntax to succinctly express the conditions desired. Property checks are used to gather relevant non-structural information which is also needed to determine if the rule transformer should be applied, such as checking whether two identifiers in the matched subtree refer to the same constant value. Figure 3 shows the rule checker for rule **C** and shows an application of some of the matching features described above.

Rule transformers always instantiate the multiplexer structure illustrated in Fig. 2, though not necessarily in the same AST location on which the rule matched. These multiplexers select an input depending on some free variables. If an assignment to these free variables is necessary to correct the design, the 2QBF solver will find the required assignment (this is described later in Sect. 3.4). Some transformers include a second type of free variable—a *pure free variable*—which can be used to correct constant values (see rule **F**). For example, the condition check $\mathtt{x} < \mathtt{y} + 3$ can be corrected to $\mathtt{x} < \mathtt{y} + 5$ with this second type of free variable.

As multiple rules may be triggered on the same AST node, we propose applying the rules following a predetermined ordering roughly going from rules that are more specific to those that are more general. Although this case does not happen with the common error library described here, Table 1 is ordered by priority (the first rule is checked/applied first).

3.3 Instrumentation of the Buggy Circuit

To implement the error rules above, we have modified the frontend of Yosys [18], an open source framework for RTL synthesis, to automatically instrument the buggy input circuit. We perform a bottom-up, depth-first traversal of the AST to trigger our code instrumentation. For each node in the traversal, we run each rule checker's structural and property checks around the AST location to identify whether there exists a rule in the common error library which can be applied. When a rule is triggered, the AST is modified to include the option of replacing or modifying the original AST with multiple potential corrections. All modifications result in additional primary inputs added to the faulty circuits: these free variables control whether the circuit retains the original erroneous behavior or is modified by some combination of changes caused by the rule transformers.

The word "combination" above is important: our technique works perfectly well to handle multiple simultaneously bugs, so long as they are each correctable with the available rules. To ensure the solver not only chooses free variables which give correct behavior, but also employ the minimal necessary number of changes, we also add an extra primary output to the instrumented design that is asserted when the number of non-zero free variables is less than some specified threshold. This threshhold is then swept, beginning with only one change allowed and ending with a user-specified maximum number of allowed changes. We arbitrarily chose a maximum threshold of three changes for our experiments.

The next step is to construct a miter: a circuit where, through the solution of a particular satisfiability problem, one can determine a concrete value for all such free variables which render the circuit correct over all inputs.

3.4 The 2QBF Problem

SAT-based combinational equivalence checking is usually performed by constructing a special circuit, called *miter*, composed of the circuit under test and a

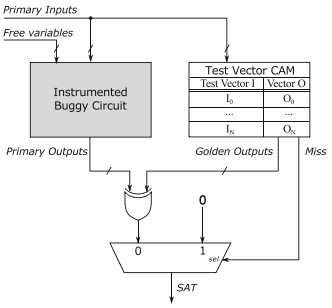

Fig. 4. Constructing a miter with test vectors. Since we have no functional reference, we build some golden outputs from a small subset of passing test vectors and all the failing test vectors we are trying to correct. The existance of a particular golden output for a given set of primary input is used to determine whether the output comparison is relevant or not.

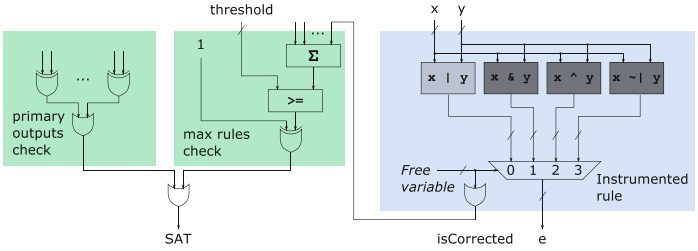

Fig. 5. Minimizing source-code interventions. The implementation of each rule transformer stores the free variables used to select a candidate change. Once all of a module's AST has been checked and transformed, these free variables are collected and their Boolean reduction is summed. The signal "isCorrected" above represents the negated Boolean reduction we actually use. A non-zero Boolean reduction (thus, a non-zero free variable) signifies that a multiplexer is configured to change the behavior of some part of the circuit. The miter then counts the number of corrections applied to the circuit and forces it to be below a fixed threshold.

functional reference circuit. In short, the solver determines if there is any input assignment which violates the expected output value of the miter. A buggy circuit will result in a satisfiable instance and any 'witness' returned is a counterexample, or error trace. Formally stated, SAT solvers determine the truth of the propositional formula:

$$\forall \boldsymbol{x} : \text{test}(\boldsymbol{x}) \Leftrightarrow \text{reference}(\boldsymbol{x}). \tag{1}$$

In contrast, syntax-guided synthesis uses SAT solvers to determine the truth of a slightly different propositional formula (formally, an *exists-forall 2QBF*) in which the circuit test also consumes some vector of free variables $\boldsymbol{h}$:

$$\exists \boldsymbol{h}\ \forall \boldsymbol{x} : \text{test}(\boldsymbol{h}, \boldsymbol{x}) \Leftrightarrow \text{reference}(\boldsymbol{x}). \tag{2}$$

This is equivalent to answering the question "does there exist some value for the special inputs $\boldsymbol{h}$ such that for all inputs $\boldsymbol{x}$, test and reference behave identically?" The internal mechanics of the solver used in this work (see Sect. 5) are beyond the scope of this paper and we do not claim any innovation on this front. Yet, we need to construct the miter in a slightly particular way given our context.

3.5 Miter Construction

Although the basic idea of the miter we use is pretty conventional for syntax-guided problems, there are two aspects which are peculiar to our situation. First, in our case we assume that a reference design is not available and that the error is exposed by an error vector or trace used for functional simulation. Second, we want to control (and thus minimize) the number of individual corrections to the buggy code.

Figure 4 shows how we build the miter from the instrumented buggy circuit and a set of simulation traces, some of which expose the error. We add an extra multiplexer at the output of the miter to ensure that the solver only tries to match the output for the given input test vectors (as opposed to any input assignment). Thus, our miter is trivially satisfied (i.e., the primary output is 0) for all input stimuli not included in the subset of simulation traces we consider. For those input stimuli which *do* match one of the simulation traces, the primary outputs of the template are XORed with the correct output response. Accordingly, our miter is satisfied by a given vector of free variables (i.e., by a specific set of error rules correcting the error) when the functionality of the instrumented circuit matches the correct output response for all input stimuli in our restricted domain. One key advantage to using this construction as opposed to a golden reference, aside from the typically-limited availability of such a golden reference, is it enhances scalability. Of course, there is a trade off between scalability and the ability of our method to find a real correction as opposed to simply turning the buggy circuit into another buggy circuit which only works correctly for the formerly failing vector and for a handful of other vectors. We will discuss later our very encouraging practical findings, largely dependent on the selective application of the error rules which we will describe soon in Sect. 4. Yet, irrespective of our positive results, one should notice two points: First, we aim to provide *meaningful* solutions to the designer and we assume that false solutions, such as those potentially produced using too few passing test vectors, would be immediately identified and discarded. Secondly, if this were not the case, it would be easy for the designer to tentatively implement the correction and verify with his or her standard verification flow if otherwise passing vectors now fail.

Besides the functional equivalence constraint, we also encode a second type of constraint to force a minimum number of corrections in the buggy RTL code. Figure 5 shows the logic responsible for this second check, mostly in the shaded area annotated as "max rules check". We simply sweep the value of the constant *threshold* in successive runs of the 2QBF solver until we find the minimum

number of changes. We are thereby able to find the minimal source code modification(s), which we intuit is/are closest to what a human would do, and try our best to rule out less general but still legal solutions.

One final type of constraint may be desirable. We do not consider multiple solutions, but they can be easily handled. At each tested threshhold value, multiple feasible solutions might exist for a couple reasons: either there is one or more false solutions caused by eschewing an exact golden reference design in favor of the test vector CAM, or there are simply multiple legitimate corrections which each require the same number of RTL changes. In either case, at each solution, the previous choice for non-zero free variables can be 'blocked', thus excluding that same combination of RTL changes, until no more solutions exist. If multiple solutions with the same number of changes are found, the user can be presented with all of them, possibly ordered by some heuristic priority.

4 Selecting Areas for "Fudging"

Applying the error models described in Sect. 3 to the complete AST of a circuit may possibly identify the right correction of the buggy circuit. Yet, both the ability to generate any possible correction and the likelyhood that the correction is the intended one may be jeopardized by this naïve implementation of our idea for a couple of reasons, both of practical and fundamental nature. First, we deliberately selected in our common error library very general rules (Sect. 3.1). This is key in capturing sufficiently broad cases which are typical of erroneous implementations: we definitely meant to be generous with our rules (for instance, as already mentioned, we imagine the library to be extended progressively with new rules as their usefulness becomes apparent). The consequence of this "generosity" is that, were we to apply every rule on every possible AST node where it can be applied, the 2QBF problem would soon become intractable even for extremely simple circuits. A second, more fundamental problem, is that an indiscriminate application of our error rules would arguably lead, in most practical cases, to multiple possible solutions, some potentially quite far (both in terms of RTL and netlist location) from the "natural" correction. We solve this issue by relying on prior work in circuit debugging and using approximate and netlist-based solutions to guide our instrumentation of the buggy specification.

4.1 SAT-Based Debugger

As Fig. 1 shows, we feed the output of a state-of-the-art debugger into FudgeFactor. This output (also called a solution) of a SAT-based debugger [15] is a set of design components (RTL blocks, RTL code) that cause the propagation of a failure. This debugger takes as input the RTL description of the design, the expected behavior of the design over a set of test vectors, and returns an over-approximate—but not necessarily precise—set of solutions (i.e., the design component where the actual error is located is within this set). We use this tool

to determine the locations on which our methodology should focus to try to correct the failure. The details of the particular SAT-based debugger we use are out of the scope of this paper and the interested reader can refer to Smith et al. [15]. All we care for is that the solution it returns is useful to the designer in most cases but contains enough ambiguity to require significant human analysis effort to lead to the actual error correction. Specifically, we parse and load the output of the SAT-based debugger and use this information to mark the corresponding AST nodes of the input circuit description as suspect. We then simply add one additional check when we implement the instrumentation pass described in Sect. 3.3: we only apply a rule checker if the node is marked as suspect.

5 Experimental Methodology

We evaluated the performance and scalability of our approach on a range of Verilog benchmarks taken from OpenCores [12]. Each benchmark has one bug either injected artificially or taken from the version control history. These buggy designs were not used to develop our common error library; they were obtained from a third party, and we do not know which bugs were injected and which are "organic". We believe our results are broadly representative of how our approach works for simple bugs in realistic circuits.

We rely on a commercial verification tool based on the work of Smith et al. [15] to obtain an initial set of error candidate locations in the input Verilog. This initial set is significantly over-approximate or, in other words, contains many false positives: most of the usually dozens of candidates are not actually part of the error. We use this set in the instrumentation process as discussed in Sect. 3.3 and unroll the resulting logic with ABC to handle sequential designs. This unrolled circuit is then passed to the CEGIS 2QBF-SAT solver [16].

Importantly, as mentioned in Sect. 3.5, we do not rely on availability of a golden reference circuit: we build a miter from only three passing test vectors. The choice of three vectors is arbitrary here, and is a trade-off between avoiding trivial, incorrect solutions, and scalability. While the topic of determining which and how many vectors to include is certainly interesting, we leave a thorough investigation to future work. The results described below appear to validate our assumption that a few test vectors are enough to properly correct most circuits with our approach: each correction found is exactly what a reasonable human designer would write, and fixes the bug most generally.

SPI (Serial Peripheral Interface) is a serial, synchronous, full-duplex communication protocol very widely used as a board-level interface between different devices such as microcontrollers, DACs, ADCs, and others. This core is an SPI/Microwire compliant master serial-communication controller with some additional functionality. There are four different buggy versions. The *bug1* design assigns the wrong signal to a control register; *bug3* contains multiple erroneous data assignments in the controller FIFO, and cannot be corrected with our restrictive threshold.

AES (Advanced Encryption Standard) is a widely used block cipher with a block size of 128 bits and a selectable key size of 128 to 256 bits. This is a pipelined 128-bit AES design from OpenCores. This core has two buggy versions. The *bug1* design is missing a subexpression in an assignment—something our methodology is poorly suited to correcting. The *bug2* design contains an XOR which erroneously references the signal used in the immediately preceding operation (a likely copy-paste error).

The *Integer Divider Core* is a parameterizable non-restoring signed-by-unsigned integer division core. In our experiments we used a 16-bit dividend and an 8-bit divisor. This design comes with seven different buggy circuits, some of which we describe here:

- *bug1.* The bug erroneously clips a signal range by one, and concatenates a two-bit constant instead of a one-bit constant. It is difficult to see how this error would be likely to occur, or how it could be corrected with a general rule.
- *bug2.* The bug is an erroneously switched set of function parameters; their order should have been reversed. Because both parameters can be changed to compatible signals, this can be corrected.
- *bug3.* In this version, the arguments of a function are both reversed, but consist of array-indexing expressions. Our rules do not capture the possibility of reversing the operands per se, although this could conceivably be corrected with another fairly-general rule.
- *bug4.* The bug references the wrong array for computing the divisor. Instead of reading the array *s_pipe*, the designer made the mistake of reading from array *d_pipe*—a typo off by one key on a keyboard.

The MIPS CPU is available on GitHub [11]. We used the CPU design to develop rules, prototype our tool flow, and validate our ability to actually solve the problems we formulate. We injected simple errors that designers commonly make and which we believe traditional debugging tools would have difficulty with: leaving out a default statement in a *case* block (in *bug1*), and mistakenly writing an *if* condition instead of an *else if* condition (in *bug2*).

6 Experimental Results

Tables 2 and 3 summarize the experimental results. The *"# Free Var. Bits"* column gives the total number of free variables used in the instrumented design (including both the control signals of all multiplexers and all pure free variables representing constants). The *"Solver Time"* column shows the cumulative solver time spent on each experiment. For example, those experiments which failed include the solver time used sequentially for all three attempted threshold values (1, 2, and 3). The *"# RTL Changes"* column describes the number of error candidate locations in the Verilog which needed fixing (for the benchmarks where a correction was found)—in other words, it is the minimum number of multiplexer free variables ("isCorrected" in Fig. 5) which need to be non-zero in

order to correct the bug. The *"Solved?"* column reports if the solver was actually able to find a solution with three or fewer changes. Note that even with our relatively sparse common error library, FudgeFactor was able to correct nearly half the simple bugs in the third-party designs.

Table 2. Those experiments listed above the break were provided by a third-party and not used to develop rules; those below the break are contrived, but show meaningful results. Note that we correct nearly half of the non-contrived experiments. Note also that all solutions are indeed those which an oracle would provide: exactly what any reasonable human designer would provide. "# Matched Nodes" lists how many AST nodes matched one (or more) of our rules. Finally, "SLOC" represented the lines of RTL source code (excluding comments, etc.).

Buggy Design	# RTL Changes	Solved?	Oracle Soln.?	Fixing Rule(s)	Applied Rules	Total AST Size	# Matched AST Nodes	SLOC
`spi_bug1`	1	✓	✓	D	ABDEF	2968	20	271
`spi_bug2`	–	×	–	–	BD	2964	2	266
`spi_bug3`	–	×	–	–	DEF	2968	10	266
`spi_bug4`	1	✓	✓	F	ABDF	2968	13	266
`aes_bug1`	–	×	–	–	ADFG	5080	19	467
`aes_bug2`	1	✓	✓	D	ABDG	5251	33	467
`div_bug1`	–	×	–	–	ADF	2486	13	163
`div_bug2`	2	✓	✓	DD	AD	2478	8	165
`div_bug3`	–	×	–	–	ADF	2486	13	165
`div_bug4`	1	✓	✓	D	ADF	2502	10	165
`div_bug5`	–	×	–	–	ADF	2516	15	168
`div_bug6`	–	×	–	–	ADF	2528	20	165
`div_bug7`	2	✓	✓	DD	ADF	2510	12	165
`cpu_bug1`	1	✓	✓	B	BDG	3842	4	530
`cpu_bug2`	1	✓	✓	C	CDEF	3846	5	531

The *"Fixing Rule(s)"* column describes which rule(s) were essential to correct the bug. In this column, we see that one rule appears with striking regularity: rule **D** (see Table 1). This should come at no surprise, as this is one of the most general rules in our library. *"Applied Rules"* lists all rules which were employed in the instrumentation phase for each experiment. Finally, *"Oracle Sol."* indicates whether the correction returned matches that which an oracle would give: if the changes were what any reasonable designer would do, we say, "yes" here. Importantly, all of the solutions found were indeed "oracle solutions". Although we do not, and will never, solve every bug, FudgeFactor reports *no* false positives while maintaining a reasonable true positive rate. We should also emphasize that the true positive rate is artificially lowered by our decision to develop the rules with only a limited set of examples and mostly based on our intuition as designers: as mentioned, we have treated all buggy designs above the

Table 3. More information on the experiments. We show the total number of free variable bits inserted, the total solver time, the size (in And-Invert gates as reported by ABC) of the associated golden reference design, the number of frames it was unrolled, and the total blowup (i.e., how much larger the instrumented circuit is than the unrolled golden reference design.

Buggy Design	# Free Var Bits	Total Solver Time (s)	# Golden Gates	Unroll Frames	Blowup
`spi_bug1`	92	1.90	14468	20	2.94x
`spi_bug2`	8	1.69	14468	20	1.20x
`spi_bug3`	35	2.23	14468	20	1.90x
`spi_bug4`	65	1.66	14468	20	2.14x
`aes_bug1`	373	18.71	86878	6	1.07x
`aes_bug2`	62	517.40	86878	6	1.29x
`div_bug1`	33	32.28	96767	48	2.30x
`div_bug2`	20	71.47	96767	48	2.12x
`div_bug3`	30	21.82	96767	48	2.28x
`div_bug4`	26	78.90	96767	48	2.24x
`div_bug5`	37	49.05	96767	48	3.20x
`div_bug6`	32	17.75	96767	48	1.99x
`div_bug7`	30	101.46	96767	48	3.15x
`cpu_bug1`	12	87.53	34294	15	2.28x
`cpu_bug2`	46	60.05	34294	15	2.56x

break as a clean test-set which has not been used to develop rules. On the other hand, in practice, the extensibility of the common error library is a fundamental part of our approach and many (but not all) of the unsolved designs could be fixed by developing additional general rule.

These tables also include some information which can be useful in determining how practical our approach is and validating our use of the SAT-based debugger to compute an over-approximate error set; our general rules would not scale if they matched many more nodes. As rule **D** shows, our strength is in using fairly general rules, but this comes at a cost: Without hints of where to look, we would be forced to use less general rules and fundamentally limit our ability to find bugs.

7 Conclusions

Since humans introduce bugs in the language they use to describe their designs, we formulated the problem of error localization and correction of a buggy RTL circuit as the problem of synthesizing a correct circuit with minimal syntactic distance from the buggy specification. To "fudge" the buggy specification into

a rich variety of possible alternate circuits, we have used an empirical library of error models that tries to capture common errors humans make. Although our rules are quite general and produce a very generous set of alternate versions, we use them sparingly by leveraging other over-approximate, better-scalable bug localization tools. We have shown, though a controlled test-set that we have not used to develop the inital set of rules, that we can correct a reasonably large set of errors and, most strikingly, in all cases we can correct, we obtain *exactly* the RTL code a human designer would have produced. As the library of common errors is extensible, we think that the success rate could be improved significantly with acceptable impact on runtimes. Nevertheless, this technique is clearly not a complete solution—it will never find all possible bugs; yet, we believe it could be invaluable in presenting, in most cases, intuitive and immediately understandable solutions to the designers and thus in freeing up precious time for them to focus on the comparatively rare hard cases where we would inherently fail.

References

1. Alizadeh, B., Behnam, P., Sadeghi-Kohan, S.: A scalable formal debugging approach with auto-correction capability based on static slicing and dynamic ranking for RTL datapath designs. IEEE Trans. Comput. **64**(6), 1564–1578 (2015)
2. Alur, R., Bodik, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Proceedings of the 13th Conference on Formal Methods in Computer-Aided Design, Portland, OR, pp. 1–8, October 2013
3. Bloem, R., Wotawa, F.: Verification and fault localization in VHDL programs. J. Telematics Eng. Soc. **2**, 30–33 (2002)
4. Chang, K.H., Markov, I., Bertacco, V.: Fixing design errors with counterexamples and resynthesis. In: Proceedings of the Asia and South Pacific Design Automation Conference, Yokohama, Japan, pp. 944–949, January 2007
5. Chang, K.H., Wagner, I., Bertacco, V., Markov, I.: Automatic error diagnosis and correction for RTL designs. In: Proceedings of the High Level Design Validation and Test Workshop, Irvine, CA, pp. 65–72, November 2007
6. Chung, P.Y., Hajj, I.N.: ACCORD: Automatic catching and correction of logic design errors in combinational circuits. In: Proceedings of the International Test Conference, Baltimore, MD, pp. 742–751, September 1992
7. Chung, P.Y., Wang, Y.M., Hajj, I.N.: Logic design error diagnosis and correction. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **2**(3), 320–332 (1994)
8. Debroy, V., Wong, W.E.: Using mutation to automatically suggest fixes for faulty programs. In: Proceedings of the Third International Conference on Software Testing, Verification and Validation, pp. 65–74, April 2010
9. Foster, H.: Trends in functional verification: a 2014 industry study. In: 2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC), pp. 1–6, June 2015
10. Jobstmann, B., Griesmayer, A., Bloem, R.: Program repair as a game. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 226–238. Springer, Heidelberg (2005)
11. Mahler, J.: A MIPS CPU written in Verilog. https://github.com/jmahler/mips-cpu. Accessed 24 April 2015
12. OpenCores: Opencores database. http://www.opencores.org

13. Peischl, B., Wotawa, F.: Automated source level error localization in hardware designs. J. IEEE Des. Test Comput. **23**(1), 8–19 (2006)
14. Singh, R., Gulwani, S., Solar-Lezama, A.: Automated feedback generation for introductory programming assignments. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, Seattle, WA, pp. 15–26, June 2013
15. Smith, A., Veneris, A., Ali, M.F., Viglas, A.: Fault diagnosis and logic debugging using Boolean satisfiability. IEEE Trans. Comput. Aided Des. Integr. Circ. Syst. CAD **24**(10), 1606–1621 (2005)
16. Lezama, A.S.: Program synthesis by sketching. Ph.D. thesis, UC Berkeley, December 2008. http://eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-177.html
17. Staber, S., Jobstmann, B., Bloem, R.: Finding and fixing faults. J. Comput. Syst. Sci. **78**(2), 441–460 (2012)
18. Wolf, C., Glaser, J., Kepler, J.: Yosys - a free Verilog synthesis suite. In: Proceedings of 21st Austrian Workshop on Microelectronics, Linz, Austria, October 2013