Towards Unifying Localization and Explanation for Automated Debugging

Görschwin Fey André Sülflow Rolf Drechsler
Institute of Computer Science, University of Bremen
28359 Bremen, Germany
E-mail: {fey,suelflow,drechsle}@informatik.uni-bremen.de

***Abstract*—Today, there exist powerful algorithms for automated debugging. Some of the debugging algorithms focus on fault localization while others try to explain the faulty behavior by providing, e.g., correct traces that are similar to a failure trace. SAT-based debugging locates faults, but does not explain the faulty behavior, e.g., some temporal properties of fault candidates are not fully explored.**

In this work, we study the resolution of SAT-based debugging with respect to its capability to locate faults and to explain faults. A strategy is presented that increases the diagnostic resolution of SAT-based debugging by combining fault localization and fault explanation in one algorithm. The experimental results confirm the strength of the approach and give directions for further research.

I. Introduction

Debugging is one of the bottlenecks in today's design flows. Verification tools and validation approaches detect misbehavior based upon a given specification. Such misbehavior is returned in terms of a *counterexample* showing a deviation from the specification. Once a counterexample is available, the following correction of this misbehavior is often done manually. As this requires expert knowledge and typically consumes a significant portion of the whole design time, automation is required.

Different automation techniques have been proposed to support the debugging step. We group these approaches in those based on explanation and those based on localization.

The explanation based approaches assist the designer in understanding why a certain execution trace of the design leads to unexpected behavior. An example is the work in [1] that compares a counterexample to similar failing and passing execution traces. The work in [2], [3] as well as [4] tries to reduce a counterexample to those parts necessary to excite a bug.

Localization based approaches are tightly linked to *model based diagnosis* that uses a formal model of the system and tries to correct this model. Already very early approaches were based on algorithms reasoning on the model [5], [6]. A correction on the model corresponds to a modification in the real system. By this, a potential location of the bug – a *fault candidate* – is found. In particular for diagnosing failing hardware efficient algorithms have been developed that help to diagnose failures (e.g. [7]) based on efficient formal reasoning engines. With the advance of very efficient solvers for *Boolean Satisfiability* (SAT) powerful approaches for SAT-based debugging of large circuits on gate level have been proposed [8]. These have been extended to handle register transfer level descriptions in *Hardware Description Languages* (HDL) [9], [10] and to debug in the context of model checking [11]. With the availability of solvers for *Satisfiability Modulo Theories* (SMT) that allow for more compact representations of SAT problems together with structural and semantic knowledge, the efficiency can be further increased [12], [13].

The accuracy of the SAT-based debugging approach can be improved in different ways. First, instead of considering a single counterexample, using multiple counterexamples helps to reduce the number of fault candidates, i.e., to localize the bug more accurately. This has been considered in the basic approach already [8]. The selection of counterexamples that improve the accuracy can be automated by using a heuristic [14] or a given specification [15], [16]. In particular for sequential circuits the resolution in the time domain can be further improved by differentiating at what time frames a correction at a certain fault candidate (e.g., a module or a gate) is required and in which other time frames no modification is required [17]. A similar analysis is performed in [18] to select signals to be stored in a trace buffer for post-silicon debugging.

In this paper, we particularly focus on sequential circuits and on bringing together localization and explanation. We consider the SAT- (and SMT-)based debugging approach in detail. We show what resolution is achieved by considering the standard approach [8] and how the resolution can be improved similarly to [17]. Different debugging strategies formalize these approaches and their relations are discussed. In particular we show how the standard strategy and a high resolution strategy can be combined to achieve a detailed diagnostic resolution while still being able to apply efficient debugging algorithms. The results obtained from diagnosing multiple counterexamples may be used to align counterexamples and to rank fault candidates. This provides helpful explanations of erroneous behavior to support the designer.

This paper is structured as follows: Preliminaries including the standard SAT-based debugging approach are revisited in Section II. Section III discusses the resolution achieved and required by a debugging approach. Moreover, different debugging strategies are defined that embed the previous SAT-based approaches. These strategies are evaluated on benchmark circuits in Section IV. Section V presents conclusions.

This work was supported in part by the European Union (Project DIAMOND, FP7-2009-IST-4-248613) and in part by the Deutsche Forschungsgemeinschaft (grant no. FE 797/6-1)

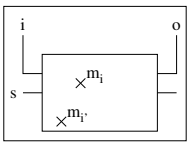


Fig. 1. Example circuit

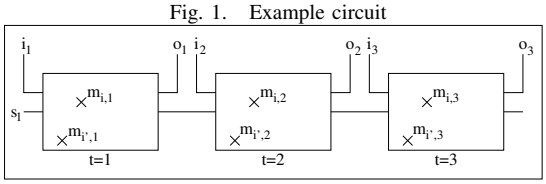


Fig. 2. Unrolling of the circuit

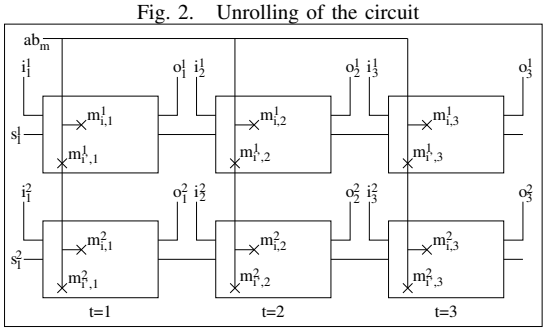
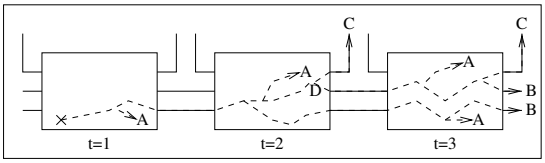


Fig. 3. Debugging instance

II. Preliminaries

A circuit $\mathcal{C}$ is composed of modules, a module is denoted by m. A module corresponds to a unique location in the HDL source code which may be a module defined in the HDL or a statement in the HDL. Each module may be instantiated multiple times. An instance i of a module m is denoted by m_i. The set $inst(m)$ contains all instances of m. An example circuit is shown in Figure 1 where module m has two instances m_i and $m_{i'}$. We consider sequential circuits over time. An instance m_i at time frame t is denoted by $m_{i,t}$. Such an unrolling (as known from formal verification) or an iterative logic array (as known from testing) is shown in Figure 2.

Counterexamples that produce incorrect output values are considered together with expected correct output values for debugging. Counterexamples provide an initial state, values for primary inputs over time, and the expected output values over time. Based on this unrolling a problem instance is created to calculate fault candidates in SAT-based debugging. Multiple counterexamples may be considered during debugging. Considering an instance m_i with respect to counterexample c at time frame t is denoted by $m_{i,t}^c$. The set of all counterexamples is denoted by *cex*. Figure 3 shows the structure of a debugging instance for the example circuit for two counterexamples. Constraining the inputs and the initial state to the counterexample and the outputs to the expected output value yields a contradiction (as the circuit produces erroneous output). Therefore *abnormal predicates* are added to the SAT instance. Let the output of an instance $m_{i,t}^c$ be the signal vector f. These output values are replaced by new outputs f' depending on the abnormal predicate $ab(m_{i,t}^c)$ as follows: $\overline{ab}(m_{i,t}^c) \rightarrow f \equiv f'$. In other words, if the abnormal predicate for $m_{i,t}^c$ is one (i.e., *active*) arbitrary values may be injected into the circuit at $m_{i,t}^c$ to rectify the counterexample. The addition of abnormal predicates is performed for all



A – masked bug
B – manifested in state
C – manifested at output
D – branching point

Fig. 4. Schematic debugging problem

instances of all modules. In the standard SAT-based debugging approach [8] all abnormal predicates belonging to instances of a single module are shared as indicated in Figure 3. Each satisfying solution to this SAT instance provides a set of activated abnormal predicates. The set of modules belonging to these abnormal predicates form a *fault candidate*. The number of these modules defines the *cardinality* of a fault candidate. Typically, fault candidates of minimal cardinality are of interest to rectify all counterexamples with the smallest modification of the source code. This is discussed in more detail in the next section.

In hierarchical debugging [9] each module itself may be composed of submodules. This can be embedded here, but is not focus of the current work.

In the following we interchangeably use Boolean conditions in sums or in Boolean expressions for brevity. If a Boolean condition is true, this is counted as 1. If a Boolean condition is false, this is counted as 0.

Instead of using SAT solvers, we apply an SMT solver [19] for debugging [12], [13]. While SAT solvers require a problem formulation in terms of a Boolean expression in conjunctive normal form, SMT solvers handle more compact constraints defined, e.g., in bit-vector logic or array theory. This allows for a more compact representation of circuits given in HDL source code, e.g., operators like a bit-vector multiplication can directly be translated into an appropriate operator for the SMT solver. The SMT solver exploits such semantic and structural knowledge to improve the efficiency over SAT solvers.

III. Debugging Resolution

In this section debugging approaches are compared with respect to their capability to locate faults and to explain faults. Section III-A motivates this work on an example. Strategies to efficiently extract temporal and spatial information are presented in Section III-B. Section III-C discusses further improvements.

A. Motivation

Consider the schematic illustration of a debugging problem shown in Figure 4. The Figure shows a circuit unrolled over three time frames. Cross references between gates in the netlist representation and HDL source code are assumed to be available. Now, assume a buggy statement generates the gates denoted by $\times$ in the first time frame and these gates are considered as a module. Upon sensitizing this module, a faulty value propagates through the circuit as indicated by the dotted lines. The faulty value may be masked as indicated by 'A', may manifest in the state of the circuit as indicated by 'B', or may be observed at primary outputs as indicated by 'C'. All modules along a path between the buggy statement and the location where erroneous output is observed may be

considered fault candidates. In the figure, the path splits up into multiple branches at the point indicated by 'D'. Behind that point, multiple locations have to be modified to cancel the effect of the bug. This effect has been exploited by early simulation-based approaches for design diagnosis already [7] and allows to use a simple path tracing procedure considering multiple counterexamples for debugging errors originating from a single location[1].

In the context of SAT-based debugging, the standard procedure assigns the same abnormal predicate to each occurrence of a module [8] in each time frame as indicated in Figure 3. Moreover, the same abnormal predicate is reused for all counterexamples in case multiple counterexamples are considered. As a consequence in the example shown in Figure 4, only a single abnormal predicate has to be activated to correct the erroneous behavior. By finding a correction where a minimal number of abnormal predicates is set to one, a minimal number of modules has to be changed to correct the erroneous behavior. But no further resolution in the time domain and in the instantiation hierarchy is available. The feedback which time frame has to be considered to correct the erroneous behavior with respect to a certain counterexample is missing. The designer has to figure out these aspects manually.

The approach of [17] improves the resolution in the timing domain - the formulation there is based on *Maximum Satisfiability* (MaxSAT), i.e., a solver that searches for the maximal subset of clauses being satisfiable. The formulation based on MaxSAT shows, that temporal information helps to explain the faulty behavior. Also using a standard SAT solver for this purpose based on a similar formulation is possible. In this case a fine grain control on the modules that are considered for correction is possible while this control is left to the MaxSAT solver in the other case. Essentially, the SAT solver provides a detailed resolution on the modifications with respect to time by using separate abnormal predicates in every time frame. Time frames where the modification of a fault candidate corrects erroneous outputs are explicitly determined. This reduces the manual effort spent on understanding the bug. The work in [18] also considers time intervals for the analysis where the goal is to decide which signals should be stored in a trace buffer for post-silicon debugging.

Finally, each instance of a module may be handled separately or all instances are handled by a single abnormal predicate. Similar to a more accurate resolution in the time domain, separately handling instances of modules also helps to understand a bug trace more easily.

B. Debugging Strategies

Using separate abnormal predicates $ab(m^c_{i,t})$ for each instance i of a module in each time frame t with respect to each counterexample c yields the most detailed resolution for debugging. This association of abnormal predicates is assumed in the following for the underlying debugging instance. By applying different types of constraints to these abnormal predicates all of the above debugging approaches can be embedded and compared with respect to their diagnostic resolution. We introduce different debugging strategies based on this observation in the following called *FINE, MODULES*, and *EXPLAIN*.

[1]In fact, the procedure in [7] considered diagnosis on the transistor level, but can be lifted to debugging HDLs by grouping gates into modules [10].

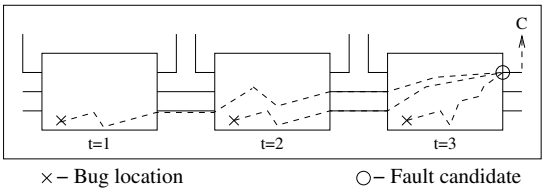


Fig. 5. Example for different strategies

FINE – To utilize all information available, all abnormal predicates are considered separately. In the problem instance this is formulated by assigning new abnormal predicates for each time frame and each instance. Then, the aim is to minimize the following sum, i.e., the total number of abnormal predicates activated for each instance in each time frame with respect to each counterexample:

$$c_F = \sum_{m \in C} \sum_{i \in inst(m)} \sum_{t \in [1,k]} \sum_{c \in cex} ab(m^c_{i,t}) \qquad (1)$$

In a satisfying solution for the debugging instance a number of c_F abnormal predicates has the value one. These correspond to a tuple $\{i_1, \ldots, i_{c_F}\}$ of module instances at certain time frames. All these tuples are collected in the set FC_F.

FINE yields the least number of modifications on the temporal model required to rectify all counterexamples. But this does not necessarily mean, that the strategy returns the least number of modules to be modified for correcting the counterexamples which can be seen as follows.

Example 1. *Assume a single bug in a design, e.g., an operator replaced by some other operator. Now, consider Figure 5 where the location of the bug is indicated by ×. The bug is excited in all time frames, but propagates to the primary output in time frame $t = 3$ in all cases. Consequently, a correction at the primary output in the third time frame would be sufficient. But as the three occurrences of the bug are handled by independent abnormal predicates, modifying the bug location would be counted as three modifications. Therefore the actual bug location is not even within the set FC_F when applying strategy FINE.*

Therefore this strategy is not suitable, if all erroneous output is assumed to depend on the same bug(s), i.e., certain locations in the source code. On the other hand, if the source of the failure does not depend on the location, e.g., in case of independent faults due to particle strikes, the strategy may provide a useful explanation of the observed behavior.

MODULES – As motivated above, one debugging goal is to modify as few modules as possible. In terms of the problem instance this is formulated by minimizing the following sum:

$$\begin{aligned} c_M = & \quad \textstyle\sum_{m \in C} ab_M(m), \text{where} \\ ab_M(m) \leftrightarrow & \quad \textstyle\bigvee_{i \in inst(m)} \bigvee_{t \in [1,k]} \bigvee_{c \in cex} ab(m^c_{i,t}) \end{aligned} \qquad (2)$$

Minimizing c_M yields the smallest number of modules that have to be modified to correct all counterexamples. Similarly, to the above strategy we collect all c_M-tuples of modules in the set FC_M. In case of $c_M = 1$ the modification of a single module is sufficient. This strategy is identical to the standard SAT-based debugging approach [8].

The strategy *MODULES* ensures that changing the behavior of all instances of a module fixes *all* counterexamples. Exactly $|inst(m)| \cdot k \cdot |cex|$ instances of m are allowed to be modified simultaneously. But *MODULES* returns no temporal information on the minimal number of modifications for particular time frames required to fix the faulty behavior.

EXPLAIN – In the second place we want to understand where and when the modifications of the modules provided by *MODULES* are required to better explain the bug. This demands a more accurate post-process for all tuples contained in FC_M.

In general, the value c_M (as retrieved by *MODULES*) is less-or-equal to the number of modifications determined by *EXPLAIN*. That is, even for a single counterexample multiple instances of a module may have to be activated at different time frames. At the same time fault candidates of minimal cardinality with respect to a single tuple $\{m_1, \ldots, m_{c_M}\} \in FC_M$ are in focus of the analysis.

To retrieve the explanation we minimize the following sum separately for each tuple $\{m_1, \ldots, m_{c_M}\} \in FC_M$:

$$c_E = \sum_{m \in \{m_1,\ldots,m_{c_M}\}} \sum_{i \in inst(m)} \sum_{t \in [1,k]} \sum_{c \in cex} ab(m^c_{i,t}) \quad (3)$$

The c_E-tuples of instances at certain time frames are collected in the set FC_E.

Strategy *EXPLAIN* is performed separately for each tuple of modules in FC_M to prevent pruning of search space. This is required according to following lemmas that consider different approaches to handle the tuples and modules contained in FC_M by two derived strategies *EXPLAIN_T* and *EXPLAIN_M*, respectively.

Lemma 1. *Strategy* EXPLAIN_T *considers all tuples in FC_M at the same time.* EXPLAIN_T *may not return at least one solution for each tuple in FC_M.*

Proof: Let FC_M contain the tuples $\{A\}$ and $\{B\}$ and assume the first fault candidate $\{A\}$ requires exactly one modification ($c_E = 1$), whereas fault candidate $\{B\}$ requires more than one modification. Consequently, the minimal cardinality with respect to $\{B\}$ is $c_E > 1$. But *EXPLAIN_T* returns all fault candidates of (minimal) cardinality $c_E = 1$ only and thus prunes fault candidates for $\{B\}$. ■

Lemma 2. *Strategy* EXPLAIN_M *considers all modules in any tuple in FC_M at the same time.* EXPLAIN_M *may return a tuple of c_E instances not corresponding to a tuple of c_M modules in FC_M since $c_E \geq c_M$.*

Proof: Let FC_M contain the tuple $\{A\}$ and $\{B\}$. Assume the debugging problem consists of a circuit having two instances of the modules A and B, respectively, a single time frame is considered, and a single counterexample is used. *EXPLAIN_M* considers the four instances of A and B as possible fault locations: A_1, B_1, A_2, and B_2. Fault candidates returned by *EXPLAIN_M* have a minimal cardinality of two. Therefore, $\{A_1, A_2\}$ and $\{B_1, B_2\}$, but also $\{A_1, B_2\}$ and $\{A_2, B_1\}$ may be returned as fault candidates. However, changing the behavior of two modules simultaneously is in contradiction to the minimal number of modules c_M to be modified as retrieved by strategy *MODULES*. ■

TABLE I
COMPARISON OF STRATEGIES

	FINE	*MODULES*	*EXPLAIN*
Abnormal predicates	per instance per time frame	per module	first per module then per instance
Fault candidates	instances	modules	instances of certain modules
Time frames	independent	all	related by instances

Another option to prevent pruning while considering all modules in tuples of FC_M simultaneously is the usage of additional constraints in the SAT instance. The additional constraints force the activation of a single module only. That is, the activation of the correction logic at a single instance of each module in the tuple $\{m_1, \ldots, m_{c_M}\} \in FC_M$ implies the deactivation of instances of all other modules $m' \in (\mathcal{C} \setminus \{m_1, \ldots, m_{c_M}\})$. If fault candidates of cardinality $c_M > 1$ (i.e., more than one module must be modified) are contained in FC_M, the set of fault candidates must be encoded into the SAT instance.

Table I summarizes the characteristics of the three debugging strategies.

C. Further improvements

The explanation may be further improved for all strategies by considering erroneous output values at particular time frames separately in a post-process. That is, fault candidates may be independent with respect to each erroneous output and to each counterexample. An additional separate analysis reveals the (in)dependency. This information is essential in case of fault candidates of cardinality larger than one, where the instances explain different erroneous behavior.

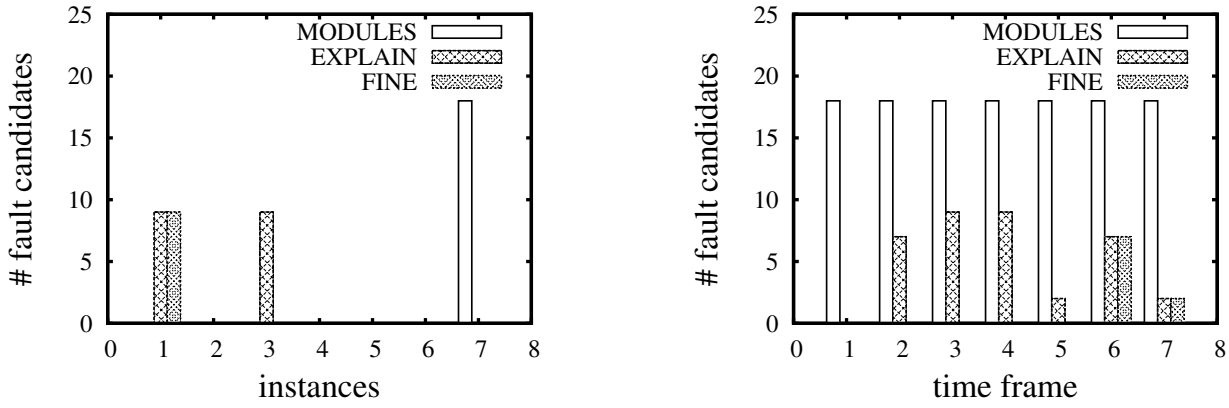
In principle, strategy *EXPLAIN* and strategy *FINE* minimize the number of active abnormal predicates separately with respect to each counterexample. Using a single SAT instance for all counterexamples requires the activation of at least one $ab(m^c_{i,t})$ in each debugging instance of a counterexample to fix the faulty behavior. The search space is large and fault candidates of cardinality $|cex|$ or larger are retrieved. Consequently, when running the SAT solver in practice, each counterexample may be handled in a separate SAT instance to reduce the search space.

For each counterexample, the time frame where a module has to be modified may be different. For example an erroneous value written to a memory may be read at an arbitrary time frame afterwards. Potential fault candidates are (1) the memory element itself while the erroneous value is stored and before it is read and (2) the real bug site producing the erroneous value before it is written. This information can be used to "align counterexamples" which further improves the explanatory capabilities of *EXPLAIN*.

Strategy *EXPLAIN* also provides a mechanism to rank fault candidates. Again consider an erroneous value stored in a memory. One potential correction in the SAT instance would be the correction of all copies of the memory in all time frames where the memory is read. Alternatively, the real bug site where the erroneous value is written to the memory needs only to be corrected once. The number of excitations of the bug, where the faulty value is propagated to a primary output

TABLE II
RESULTS

benchmark	|C|	MODULES |FC_M|	c_M	|TF|	EXPLAIN |FC_E|	c_E	|TF|	%pruned	freq.	FINE |FC_F|	c_F	|TF|	%pruned	freq.
b12	1,171	18	1.00	7	18	2.00	6	14.29	2.00	9	1.00	2	71.43	1.00
b14	10,481	140	1.00	5	149	1.05	5	0.00	1.12	141	1.00	5	0.00	1.07
cordic_p2r	3,528	44	1.00	10	48	1.00	3	66.67	1.09	48	1.00	3	66.67	1.09
i2c	1,539	7	1.00	5	7	1.00	2	60.00	1.00	7	1.00	2	60.00	1.00
rsencoder	6,057	67	1.00	10	67	1.00	2	80.00	1.00	67	1.00	2	80.00	1.00
usb	7,187	47	1.00	3	47	1.00	3	0.00	1.00	47	1.00	3	0.00	1.00



(a) Cardinality

(b) Distribution of fault candidates

Fig. 6. Details for benchmark b12

gives a minimal number of instances at different time frames, where a correction is required (e.g., multiple erroneous writes to the memory are also possible).

An alternative combination of strategy *MODULES* and strategy *EXPLAIN* in one algorithm may further speed up the classification. In this case constraints to determine c_M are added to the debugging instance, a solution minimizing c_M is calculated, and afterwards the value of c_M is fixed to this minimum. This ensures that only valid solutions with respect to strategy *MODULES* are considered. Next, constraints to determine c_E are added and solutions with increasing cardinality for c_E are calculated. This approach uses only a single debugging instance, but handles all counterexamples at the same time instead of independently.

IV. Experimental Results

This section provides experimental results to the debugging strategies introduced above. Designs from the ITC'99 benchmark suite and from OpenCores (http://www.opencores.org) are used for the experiments. Single bugs like operator replacements are randomly injected at the module level. Counterexamples are found by sequential equivalence checking of primary outputs for up to ten times frames. Modules on the word level are considered, i.e., a module may be a single gate, but also a multiplier. All experiments have been conducted on an AMD Athlon X2 processor (3 MHz, 4GB RAM) using the verification environment WoLFram [20] and the SMT solver Boolector [21].

The first series of experiments in Section IV-A focuses on the accuracy of the different debugging strategies. Section IV-B discusses the quality with respect to multiple counterexamples.

A. Explanation

Table II presents experimental results for the different strategies with respect to a single counterexample. The first two columns give the name of the benchmark and the number of modules, respectively. The accuracy is measured by the number of fault candidates (|FC_M|, |FC_E|, |FC_F|), the cardinality (c_M, c_E, c_F), and the number of time frames with active abnormal predicates (|*TF*|). Column *%pruned* compares the reduction of time frames to the number of time frames returned by strategy *MODULES*. Finally, Column *freq.* highlights the frequency of fault candidates in a time frame, i.e., a value of two means that modules have to be activated in two different time frames on average.

Focusing on the cardinality of fault candidates, *MODULES* returns fault candidates of cardinality $c_M = 1$, i.e., fault candidates that are allowed to be activated at all time frames to fix the counterexample. An analysis by *EXPLAIN* shows that often the modification of a module at one specific time frame is sufficient to fix the counterexample. Only a few fault candidates returned by *MODULES* require modifications at multiple time frames (see, e.g., b12 and b14). This confirms the observation of [17].

Strategy *FINE* is less accurate in comparison to *EXPLAIN* and often *FINE* does not return all equivalent fault candidates. For example, *FINE* misses 50% of the fault candidates for benchmark b12, i.e., modules that have to be modified in multiple time frames are pruned.

Figure 6 shows details for benchmark b12 for a counterexample of seven time frames. Fault candidates returned by *MODULES* are activated at all seven time frames. *EXPLAIN* differentiates the fault candidates in FC_M and shows $c_E = 1$ or $c_E = 3$ for all tuples in FC_M. The combination of

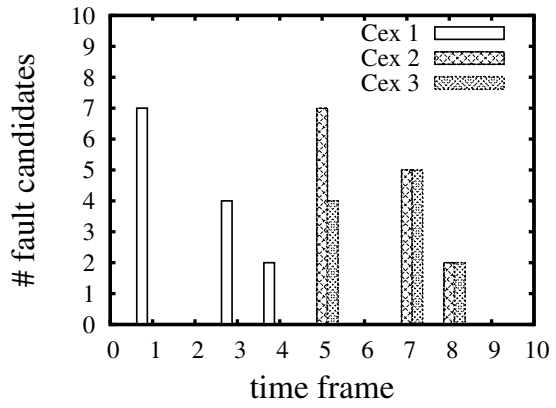


Fig. 7. Multiple counterexamples

Figure 6(a) and Figure 6(b) explains that fault candidates between time frame two and time frame five have cardinality three, whereas fault candidates in time frame six to seven have cardinality one only. *FINE* misses all fault candidates of cardinality three that have to be activated in time frame two to five.

B. Multiple Counterexamples

Multiple counterexamples further explain the erroneous behavior of a design. Figure 7 presents details for an analysis of benchmark b14 by strategy *EXPLAIN* for multiple counterexamples. The number of fault candidates in FC_E that activate an instance in a particular time frame is shown separately for each counterexample. Using three counterexamples, the set of modules is reduced compared to the results in Table II.

Often the activation of a single instance is sufficient to fix the counterexamples, i.e., $c_E = 1$ in most cases. Moreover, the counterexamples require an activation of fault candidates in different time frames. Thus, different behavior for sensitizing and observing the bug is produced by each counterexample.

However, the distribution of fault candidates with respect to the counterexamples is similar. The similar signature leads to the conclusion that the bug is the same, but with respect to the time there is an offset between the counterexamples. These results can be used to align counterexamples, i.e., to match equivalent fault candidates that are moved by an offset of t' time frames.

Non-alignable fault candidates distinguish the counterexamples and are more likely not to be the "real" candidate fault site. This helps a designer to focus the manual analysis of a counterexample to specific time frames and to specific fault candidates. Additionally, the information can be used to automatically rank fault candidates in FC_E.

In summary, the strategy *EXPLAIN* in one algorithm increases the accuracy of SAT-based debugging significantly by providing information on the excitation of fault candidates. Thus, a designer knows where (which instance) and when (which time frame) a fault candidate has to be modified which speeds up debugging.

V. Conclusions and Discussion

This work discussed strategies to debug sequential circuits. Temporal information and spatial information have been shown to be valuable to debug a design. Moreover, the analysis of multiple counterexamples provides information on similarities and differences of counterexamples. Passing this knowledge to the designer improves the explanatory capabilities of automatic debugging.

In focus of future work is the automation of the analysis to provide better explanations to a designer. For example, the alignment of counterexamples may be automated to highlight similarities and differences of counterexamples. Moreover, the alignment can be used to automatically rank fault candidates.

References

[1] A. Groce and W. Visser, "What went wrong: Explaining counterexamples," in *Model Checking of Software: SPIN Workshop*, ser. Lecture Notes in Computer Science, no. 2648, 2003, pp. 121–135.

[2] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[3] A. Groce, "Error explanation with distance metrics," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 2988, 2004, pp. 108–122.

[4] K. Ravi and F. Somenzi, "Minimal assignments for bounded model checking," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 2988, 2004, pp. 31–45.

[5] R. Reiter, "A theory of diagnosis from first principles," *Artificial Intelligence*, vol. 32, pp. 57–95, 1987.

[6] J. de Kleer and B. Williams, "Diagnosing multiple faults," *Artificial Intelligence*, vol. 32, pp. 97–130, 1987.

[7] A. Kuehlmann, D. I. Cheng, A. Srinivasan, and D. P. LaPotin, "Error diagnosis for transistor-level verification," in *Design Automation Conf.*, 1994, pp. 218–224.

[8] A. Smith, A. Veneris, M. F. Ali, and A.Viglas, "Fault diagnosis and logic debugging using boolean satisfiability," *IEEE Trans. on CAD*, vol. 24, no. 10, pp. 1606–1621, 2005.

[9] M. Ali, S. Safarpour, A. Veneris, M. Abadir, and R. Drechsler, "Post-verification debugging of hierarchical designs," in *Int'l Conf. on CAD*, 2005, pp. 871–876.

[10] G. Fey and R. Drechsler, "Efficient hierarchical system debugging for property checking," in *IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems*, 2005, pp. 41–46.

[11] G. Fey, S. Staber, R. Bloem, and R. Drechsler, "Automatic fault localization for property checking," *IEEE Trans. on CAD*, vol. 27, no. 6, pp. 1138–1149, 2008.

[12] A. Sülflow, G. Fey, and R. Drechsler, "Experimental studies on SMT-based debugging," in *IEEE Workshop on RTL and High Level Testing (WRTLT)*, 2008, pp. 93–98.

[13] S. Mirzaeian, F. Zheng, and K.-T. Cheng, "RTL error diagnosis using a word-level SAT-solver," in *Int'l Test Conf.*, 2008, pp. 1–8.

[14] G. Fey and R. Drechsler, "Finding good counter-examples to aid design verification," in *ACM & IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, 2003, pp. 51–52.

[15] A. Sülflow, G. Fey, C. Braunstein, U. Kühne, and R. Drechsler, "Increasing the accuracy of SAT-based debugging," in *Design, Automation and Test in Europe*, 2009, pp. 1326–1332.

[16] A. Sülflow, G. Fey, and R. Drechsler, "Using QBF to increase accuracy of SAT-based debugging," in *IEEE International Symposium on Circuits and Systems*, 2010, pp. 641–644.

[17] Y. Chen, S. Safarpour, J. M. Silva, and A. Veneris, "Automated design debugging with maximum satisfiability," *IEEE Trans. on CAD*, vol. 29, no. 11, pp. 1804–1817, 2010.

[18] Y.-S. Yang, N. Nicolici, and A. Veneris, "Automated data analysis solutions to silicon debug," in *Design, Automation and Test in Europe*, 2009, pp. 982–987.

[19] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "DPLL(T): Fast decision procedures," in *Computer Aided Verification*, ser. LNCS, vol. 3114, 2004, pp. 175–188.

[20] A. Sülflow, U. Kühne, G. Fey, D. Große, and R. Drechsler, "WoLFram – a word level framework for formal verification," in *International Symposium on Rapid System Prototyping (RSP)*, 2009, pp. 11–17.

[21] R. Brummayer and A. Biere, "Boolector: An efficient SMT solver for bit-vectors and arrays," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 4963. Springer, 2009, pp. 174–177.