Using Unsatisfiable Cores to Debug Multiple Design Errors

Andre Sülflow[1] Görschwin Fey[1,2] Roderick Bloem[3] Rolf Drechsler[1]

[1] *Institute of Computer Science*
University of Bremen
28359 Bremen, Germany
{*suelflow,fey,drechsle*}
@informatik.uni-bremen.de

[2] *VLSI Design & Education Center*
University of Tokyo
Tokyo, 113-8656, Japan

[3] *Institute for Software Technology*
Graz University of Technology
8010 Graz, Austria
rbloem@ist.tugraz.at

ABSTRACT

Due to the increasing complexity of today's circuits a high degree of automation in the design process is mandatory. The detection of faults and design errors is supported quite well using simulation or formal verification. But locating the fault site is typically a time consuming manual task. Techniques to automate debugging and diagnosis have been proposed. Approaches based on *Boolean Satisfiability* (SAT) have been demonstrated to be very effective.

In this work debugging on the gate level is considered. Unsatisfiable cores contained in a SAT instance for debugging are used (1) to determine all suspects, and (2) to speed-up the debugging process. In comparison to standard SAT-based debugging, the experimental results show a significant speed-up for debugging multiple faults.

Categories and Subject Descriptors

B.6.3 [**Hardware**]: LOGIC DESIGN—*Design Aids*

General Terms

Design, Verification

Keywords

SAT-based debugging, Fault Localization, Unsatisfiable Core

1. INTRODUCTION

Circuit design happens under an enormous time-to-market pressure while the complexity of the designs increases rapidly. Tool automation in the design process is mandatory. Verification is one of the bottlenecks of today's design flows, even though the detection of faulty behavior is automated by the use of simulation, formal methods, or a combination of both. Up to now the subsequent debugging step lacks tool support.

Several automatic approaches to aid debugging have been proposed. Some of them rely on simulation [19] or structural properties of the debugging problem [9], others use strong reasoning engines [5]. Due to the drastic improvements in deciding *Boolean Satisfiability* (SAT) using powerful algorithms [10, 11, 2], SAT-based debugging is quite efficient. The first approach [16] has been further improved to exploit the hierarchy of a design [3], to debug formal properties [18] and to provide corrections [1]. Recently, performance improvements have been reported by using an engine that finds all maximal subsets of satisfiable clauses [17]. This work is related to our approach, but only single faults have been considered in [17].

In principle all SAT-based approaches rely on the same underlying model. Initially, a number of failure traces and an expected correct output response for each trace are given. Then, the circuit is transformed into a SAT instance, the primary inputs are constrained to the values given by a failure trace and the outputs are restricted to the correct output value. Obviously, this SAT instance is unsatisfiable because the failure trace implies incorrect output values. Therefore, additional *injection logic* is added to change the values of internal signals and by this to correct the output response.

Another approach [13] proposes to use conflicting sets, i.e. sets of assumptions that cannot be true simultaneously. A fault candidate contains at least one assumption of each conflicting set. Given an initial conflicting set, Reiter's approach proceeds by enumerating all assumptions in the conflicting set and extending them to a fault candidate. Whereas Reiter's approach calls the proof engine for each assumption in the conflicting set, our approach has the decisive advantage of only requiring as many calls to the SAT solver as there are components in the fault candidate (plus one call for every further fault candidate).

In this work debugging on the gate level is considered. We start from the initial unsatisfiable SAT instance: the circuit is restricted to the failure trace and to correct output values [13]. A SAT solver is applied to extract an unsatisfiable core, i.e., a part of the SAT problem that is already unsatisfiable [4, 21, 14]. Then, the injection logic is activated within the unsatisfiable core to determine fault candidates or a next unsatisfiable core. Using an iterative procedure, multiple faults are handled. The solution space of our method is identical to the standard approach for SAT-based debugging: all fault candidates of minimal cardinality are returned. Moreover, the experimental results show that multiple faults can be handled very efficiently which is difficult for previous approaches.

The paper is structured as follows: Preliminaries on SAT and SAT-based debugging are revisited in the next section. Then, the basic idea of using unsatisfiable cores is introduced in Section 3, optimizations are presented and the solution space is analyzed. Our framework that implements the debugging procedure is explained in Section 4. Experimental results are reported in Section 5. Finally, conclusions are presented.

2. PRELIMINARIES

2.1 Unsatisfiable Cores

A Boolean formula in *Conjunctive Normal Form* (CNF) is a set of clauses, each clause is a set of literals and a literal is a variable or its negation. A given CNF is satisfied under an assignment, if each clause is satisfied; a clause is satisfied if at least one of its

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GLSVLSI'08, May 4–6, 2008, Orlando, Florida, USA.
Copyright 2008 ACM 978-1-59593-999-9/08/05 ...$5.00.

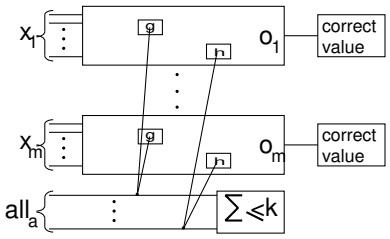


Figure 1: SAT instance

literals is satisfied; a literal is satisfied if it evaluates to 1 under the current variable assignment. A SAT solver proves that a given CNF is unsatisfiable or provides a satisfying assignment.

If a CNF is unsatisfiable, the formula is contradictory. Any subset of the clauses that is still unsatisfiable is called an *unsatisfiable core*, also called *unsat core* or simply *core* in the following. For example, the CNF formula $\{\{a\}, \{\overline{a}\}, \{a, b\}\}$ is unsatisfiable; the subset $\{\{a\}, \{\overline{a}\}\}$ is an unsat core. A given CNF may contain multiple unsat cores. An unsat core is *minimal* if the removal of one more clause yields a satisfiable CNF. Determining a minimal unsat core is possible [12, 6, 8], but time consuming. For our purpose minimality is not necessary. A SAT solver can produce such an unsat core on the fly with a moderate overhead [14].

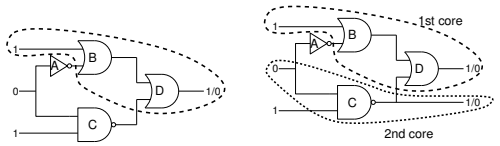
An unsat core may become satisfiable by modifying its clauses. We refer to this removal of the contradiction as *breaking* the unsat core.

2.2 SAT-based Debugging

In the following, $\mathcal{G}$ denotes a circuit and $\mathcal{X}$ denotes a set of failure traces with corresponding correct output responses. This is the input to create a SAT instance for the debugging problem [16]. Figure 1 shows the structure of the SAT instance. The circuit is replicated for each failure trace; the inputs are restricted to the values provided by the failure trace $x_1 \ldots, x_m$; the outputs are restricted to the correct values. Additional logic is used to change values of internal signals: the Boolean function g of a gate is replaced by $\overline{a}_g \rightarrow g$, i.e., if $a_g = 0$, the gate works as usual; if $a_g = 1$, the output value of the gate is unrestricted. Therefore, if $a_g = 1$ and the gate is faulty, the SAT solver can choose the correct output for the gate. The variables a_g are called *abnormal predicates* and all_a denotes the set of all abnormal predicates. As shown in Figure 1, the same abnormal predicate is used for a gate with respect to all failure traces. We limit to k the total number of abnormal predicates set to 1. Each satisfying assignment to this SAT instance yields a fault candidate containing k gates. All gates with a_g set to 1 are contained in the fault candidate.

Typically the algorithm runs in a loop that starts with $k = 1$ to identify single gates as fault candidates. If no solution is found, k is incremented until the first candidate is determined. By inserting a blocking clause for a candidate, all candidates of size k are calculated [16]. In the following we refer to this procedure as *standard SAT-based debugging*.

The extension to the sequential problem is straightforward: the circuit is unrolled and the same abnormal predicate is used for a gate in all time steps [16]. Similarly, instead of gates more complex components like modules or RT level expressions can be considered, by using a single abnormal predicate for all gates belonging to the same component [3, 18]. In the following our extensions to the basic algorithm are explained on the gate level for the combinational case – the extension to hierarchical sequential problems is straightforward. Experimental results include the sequential case as well.



(a) Single fault (b) Double fault

Figure 2: Fault diagnosis

3. USING UNSATISFIABLE CORES

This section explains the basic procedure of using unsat cores for SAT based debugging:

1. Create the debugging instance and force all abnormal predicates to 0.
   (a) Calculate an unsat core.
   (b) Break the core by activating the abnormal predicates of gates in the core.
   (c) If the problem is still unsatisfiable, go to (a), else go to 2.
2. Do standard SAT-based debugging.

3.1 Single Faults

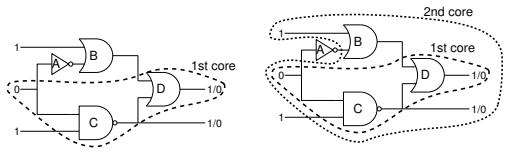
The idea behind the procedure is the following. First consider the case of a single fault. As explained earlier, the SAT instance for debugging is unsatisfiable when all abnormal predicates are set to 0 – the failure trace implies erroneous values at outputs, therefore constraining the outputs to correct values yields a contradiction. This contradiction is due to the fault that is present in the circuit: if the fault is corrected, the correct output values are implied by the failure trace.

This argumentation applies to the CNF level as well: The initial SAT instance contains an unsat core. Clauses derived from the actual fault site must be part of this unsat core. Therefore only those gates that share clauses with the unsat core have to be considered during debugging. Only abnormal predicates of such gates are activated, i.e., allowed to take the value 1 – by this the core is broken. This reduces the search space for the SAT engine as only a small subset of the abnormal predicates is active. In the following we refer to the set of activated abnormal predicates as the suspects $\mathcal{S}$. These are considered in the second unsat core debugging step (see above) during standard SAT-based debugging. This step finds all fault candidates within $\mathcal{S}$.

EXAMPLE 1. *Consider the circuit shown in Figure 2(a). Under the input assignment* $(1, 0, 1)$ *the output takes the value* 1 *while a* 0 *would be the correct value. The dashed shape encloses an unsat core. Therefore the suspects for SAT-based debugging are* $\mathcal{S} = \{B, D\}$*, and* C *is not a candidate. Allowing one suspect to behave "abnormal" makes the problem satisfiable; gate* D *is a fault candidate. In contrast, gate* B *is not a candidate.*

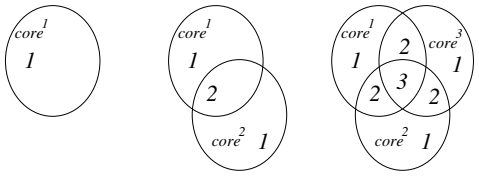
Note that multiple unsat cores may be present even in case of a single fault, a single failure trace and a single output that is considered. Calculating all of these cores and intersecting them would help to further reduce the suspects [13]. On the other hand, all of these cores are broken if the abnormal predicate of the fault site is set to 1. Thus, determining a single core per fault is sufficient to apply standard SAT-based debugging. Calculating multiple unsat cores may be expensive.

The first unsat core can be derived very efficiently. The SAT solver only performs implications (using Boolean constraint propagation [10]) which yields the contradiction. Thus, no decisions are necessary to retrieve the core.



(a) First core (b) Second core

Figure 3: Double fault – alternative cores



(a) 1st core (b) 2nd core (c) 3rd core

Figure 4: Multiple fault debugging

3.2 Multiple Faults

Now, consider multiple faults. Even after determining the first unsat core and activating abnormal predicates, the SAT instance may remain unsatisfiable due to another fault. Thus, the next unsat core is calculated and the abnormal predicates of the corresponding gates are activated, i.e., the set of suspects $\mathcal{S}$ is extended. This process is iterated until no unsat cores remain.

EXAMPLE 2. *Consider the circuit shown in Figure 2(b). The circuit is similar to the one used in the previous example, but it has one more output, which is also incorrect. With respect to the first output, the same unsat core is extracted as previously, i.e.,* $\mathcal{S} = \{B, D\}$*. After allowing up to one suspect to behave abnormally, the formula is still unsatisfiable: the new output cannot be corrected. The unsat core of this call is* $\{C\}$ *and the new set of suspects is* $\mathcal{S} = \{B, C, D\}$*. Now, we allow two of these subjects to behave abnormally. This time, all unsat cores can be broken. The pairs* $\{D, C\}$ *and* $\{B, C\}$ *are fault candidates.*

This leads to the generalization of the constraints to limit the abnormal predicates. Assume that altogether k cores were iteratively determined. The set $core^i$ contains all abnormal predicates of gates in core i, i.e., there are k such sets $\{core^1, \ldots, core^k\}$. Formally, the suspects are given by $\mathcal{S} = \cup_{i=1}^{k} core^i$. The following constraints are embedded in the debugging instance:

1. At least one abnormal predicate within each unsat core must be 1 in order to break the core:

$$|\{a \in core^i \mid a = 1\}| \geq 1, \quad \text{for } 1 \leq i \leq k \tag{1}$$

2. The total number of abnormal predicates with value 1 must equal the number of cores k to retrieve the fault candidate of smallest cardinality:

$$|\{a \in \mathcal{S} \mid a = 1\}| = k \tag{2}$$

3.3 Optimizations

The search space grows exponentially in presence of multiple faults [13, 19]. But using unsat cores helps to analyze the dependencies between the faults and as a consequence to prune the search space.

EXAMPLE 3. *Again consider the circuit shown in Figure 2(b). Two non-overlapping unsat cores are calculated leading to two sets of abnormal predicates* $core^1$ *and* $core^2$*. The constraints as defined in Equations (1) and (2) in principle yield* $\binom{|\mathcal{S}|}{2}$ *possibilities to choose the values of abnormal predicates. Knowing that the cores are non-overlapping and that one abnormal predicate per core must be* 1*, only* $|core^1| \cdot |core^2|/2$ *of these possibilities have to be considered. By additional constraints this information can directly be encoded to guide the SAT solver. While irrelevant for the example, this observation becomes important for large values of the number of faults* k *and suspects* $|\mathcal{S}|$*.*

Thus, multiple non-overlapping cores can be treated independently. As a first refinement, Equations (1) and (2) are replaced by the following constraints for non-overlapping unsat cores $\{core^1, \ldots, core^k\}$ — exactly one abnormal predicate per core must be 1:

$$|\{a \in core^i | a = 1\}| = 1, 1 \leq i \leq k \tag{3}$$

For overlapping cores these constraints would be too tight and prune candidates as the following example shows.

EXAMPLE 4. *The same circuit is considered again, but different unsat cores are used as shown in Figure 3. The first unsat core contains a contradiction with respect to the expected value of the upper output (see Figure 3(a)). Activating the abnormal predicates in this core and allowing at most one of them to become* 1 *does not yield a correction for both outputs: if gate* D *is changed, only the upper output is corrected; if gate* C *is changed, only the lower output is corrected. Thus, a next unsat core is returned as shown in Figure 3(b). This core completely contains the first core due to the following contradiction: Repairing the lower output requires* $a_C = 1$ *which implies* $a_D = 0$ *(so far only one unsat core has been found, therefore* $\sum_{a \in \mathcal{S}} a$ *is restricted to be less or equal* 1*); thus the upper output cannot be corrected due to the controlling value* 1 *at* D *coming from* B*.*

Obviously, the process depends on the order in which the unsat cores are found, e.g., this determines how many abnormal predicates have to be considered during the standard SAT based debugging process. A heuristic could be used to guide this process. But in practice, it is expensive to determine multiple unsat cores while producing a first one is a side effect of the solving process. Therefore, we use the first core determined during a run of the SAT solver.

The previous example also shows that the knowledge about the intersections of the unsat cores can be utilized to further prune the search space in presence of multiple faults. Therfore Equation 3 can be generalized to handle overlapping cores as well.

One abnormal predicate per core must assume the value 1 to resolve the contradiction. If multiple cores intersect, both faults may be contained in the intersection. Therefore more abnormal predicates are allowed to take the value 1 in the intersections of cores. This is illustrated in Figure 4. At first, core $core^1$ is retrieved. The 1 in this set denotes that only one abnormal predicate in $core^i$ may be 1. Then, a next core is $core^2$ is retrieved (see Figure 4(b)). At least one gate per core must behave "abnormal" and both of these may be contained in the intersection $core^1 \cap core^2$. Therefore the number of abnormal predicates with value 1 is limited to 2 in $core^1 \cap core^2$ and to 1 in the remainder of both cores. Additionally, an overall limit as defined in Equation 2 is applied. The next unsat core $core^3$ may intersect with all previous cores. This leads to the limits indicated in Figure 4(c). Thus, the limits can be tightened for all intersections leading to a partitioning of $\mathcal{S}$. Moreover, all of these limitations can be calculated efficiently incrementally during the first stage of the diagnosis procedure. On the other hand for k cores, the number of sets in the partition may be 2^k.

The different types of constraints have been implemented and experimentally evaluated. Applying individual limitations for each intersection did not yield a speed-up in general. Only for some instances with large k, speed-ups were achieved. Results are reported for the limitations provided in Equations (1), (2) and (3).

3.4 Solution Space

Another important question is the relation between the proposed framework and standard SAT-based diagnosis with respect to the solution space. Typically, fault candidates of minimal cardinality are of interest, because they are a minimal set of gates that have to be considered for correcting the circuit. Most of the previously proposed automated debugging procedures including the standard SAT-based procedure return such fault candidates.

THEOREM 1. *Fault candidates of minimal cardinality are provided.*

PROOF. Assuming that our framework creates a non-minimal fault candidate leads to a contradiction.

The standard SAT-based debugging procedure returns fault candidates of minimal cardinality by iteratively incrementing the number of abnormal predicates with value 1. Now assume that this procedure returns a fault candidate $L = \{a_1, \ldots, a_l\}$ of cardinality l and the algorithm introduced here generates k unsat cores $core^1, \ldots, core^k$ where $k > l$ until the instance becomes satisfiable and a fault candidate of cardinality k is returned.

Note that L must share at least one element with every unsat core contained in the debugging instance – otherwise the instance would still be unsatisfiable.

Now, consider the first l cores $core^1, \ldots, core^l$ returned by the algorithm. At least one of these cores does not share any abnormal predicate with L – otherwise the problem would be satisfiable at that point and the algorithm would stop. This is a contradiction to L being a fault candidate. ■

Additionally, the proposed framework should provide the same fault candidates as determined by the standard SAT-based diagnosis algorithm.

THEOREM 2. *All fault candidates of minimal cardinality are provided.*

PROOF. Any fault candidate $K = \{a_1, \ldots, a_k\}$ of minimal cardinality breaks all unsat cores, i.e., resolves all contradictions. Therefore, each core derived during the run of the framework must share at least one abnormal predicate with K. Since the set of suspects $\mathcal{S}$ is extended until all unsatisfiable cores are broken, $K \subseteq \mathcal{S}$. Moreover, each $a_i \in K$ must be contained in one core. Otherwise, two cores must be broken by the same abnormal predicate and the algorithm would have stopped after $k-1$ iterations, returning a smaller fault candidate. Thus, the limitations cannot block fault candidate K. ■

In the next section the debugging framework is introduced in more detail.

4. DEBUGGING FRAMEWORK

In the following, our implementation is presented without the optimizations of Section 3.3 for simplicity. The framework proceeds in the two stages introduced in the previous section:

1. Reduce the suspects and determine their limitations by using unsat cores (Figure 5).
2. Extract all candidate fault sites (Figure 6).

The first stage starts with the generation of the CNF formulation of the debugging problem according to Section 2.2 (line 2). The

```
1  function unsatCoreProcessing(G, X) {
2    (cnf, all_a)= createDebugInstance(G, X)
3
4    k=0;
5    S = ∅;
6    cnf.addClause "∀a ∈ all_a : a = 0";
7    do {
8      result = cnf.solve();
9      if (result == sat) break;
10
11     k++;
12     core = {a | a ∈ cnf.unsatCore() and
           a ∈ all_a};
13     cnf.addClause "|{a ∈ core|a = 1}| >= 1";
14
15     cnf.removeClause "|{a ∈ S|a = 1}| = k-1";
16     cnf.removeClause "∀a ∈ (all_a \ S) : a = 0";
17
18     S = S ∪ core;
19     cnf.addClause "|{a ∈ S|a = 1}| = k"
20     cnf.addClause "∀a ∈ (all_a \ S) : a = 0";
21   } while (true)
22
23   return (cnf, S);
24 }
```

Figure 5: First stage

CNF formula *cnf* and the set of all abnormal predicates all_a are returned. The number of fault sites (k) and the suspects ($\mathcal{S}$) are initialized (lines 4–5). Initially the number of abnormal predicates set to 1 is forced to 0 (line 6). Therefore the instance is unsatisfiable. Next, unsat cores are iteratively calculated (lines 7–21).

If the formula is satisfiable, the loop stops and the framework proceeds to the second stage (lines 8–9). Otherwise, there is another fault site and k is incremented (line 11).

The abnormal predicates in the current unsat core are extracted and limited according to Equation (1) (lines 12–13). As explained in Section 2.1 the core can be determined with an acceptable overhead during the SAT run. Afterwards the new abnormal predicates are added to $\mathcal{S}$ (line 18).

During each iteration the abnormal predicates are limited according to Equation (2). The limitation is removed after each iteration (line 15) to be replaced by an updated one (line 19). Additionally, the abnormal predicates not in $\mathcal{S}$ have to be forced to 0 (line 20) and to be removed during each iteration (line 16). This prevents the "faulty" behavior and can be implemented by using unit clauses, too. By using groups of clauses, parts of the CNF are reused and some learned information can be kept for subsequent runs [15, 20].

The loop starts again with new limitations on the fault site and the next unsat core is calculated.

When the first satisfying solution is found (line 9), the second stage starts. At this point the limits on the number of abnormal predicates are contained in the CNF formula. Therefore all candidate fault sites can be determined as proposed in [16] (Figure 6).

5. EXPERIMENTAL RESULTS

This section reports experimental results for our framework. Debugging multiple faults is much harder than debugging single faults due to the exponentially increasing search space [19]. For single faults only minor improvements are expected, due to the extra overhead of the unsat core extraction. Therefore we concentrate our analysis on multiple faults.

All experiments were carried out on a Intel Core 2 Duo (2.33

```
1  function getCandiateFaultSites(cnf, S) {
2    /* Determine all candidates */
3    candidates= ∅;
4    while (result==sat) {
5      newCandidate= {a | cnf.assignment(a)=
         1};
6      cnf.addBlockingClause(newCandidate);
7      candidates = candidates ∪ {
         newCandidate};
8      result= cnf.solve();
9    }
10   return candidates;
11 }
```

Figure 6: Second stage

GHz, MacOS 10.4, 2GB) within a run time limit of 5 hours and a memory limit of 1GB. Combinational and sequential circuits from the LGSynth93 and the ITC-99 benchmark sets were considered.

The debugging framework does not require any assumption on the fault model. For the experiments gate replacement faults were considered. Injecting n faults of this type guarantees that there exists a fault candidate of cardinality n. Note that fault masking may occur: n faults are present but a fault candidate containing $k < n$ gates is determined. This typically happens in other diagnosis frameworks as well because fault candidates of minimal cardinality are of interest. The failure traces were created by checking equivalence between the faulty circuit and the original, correct circuit. Each faulty output was covered by at least one counterexample to achieve a good resolution during debugging.

In the first two series of experiments we compare standard SAT-based debugging [16] to our framework. In [16] improvements were achieved by using structural dominators and incrementally considering more and more counterexamples. We do not exploit the presence of structural dominators. Furthermore, incrementally using more and more counterexamples is not possible when there are multiple faults: using only a few counterexamples to rule out suspects from the consideration may prune fault candidates that can only be detected by using all of the counterexamples. Therefore all counterexamples were considered simultaneously for the standard approach. Both algorithms are implemented within our framework.

The framework of Section 4 was extended by the optimization with respect to pairwise non-overlapping unsat cores (Section 3.2, Equation (3)). The additional optimization using intersections between cores is considered later. The framework was implemented on the base of Zchaff [11], which also handles groups of clauses. MiniSat [2] was not considered due to the lack of incremental SAT processing and unsat core extraction in the downloadable version. After the first stage, we have re-created the debug SAT instance for the second stage that includes the abnormal predicates for the given suspects only[1]. Afterwards, the limitations on the abnormal predicates as determined during the first stage were added.

Table 1 and Table 2 report results for combinational and sequential circuits, respectively. The columns $circuit$, $\#g$ and $\#c$ give the name of the circuit, number of gates and the number of counterexamples simultaneously considered, respectively. Column k shows the minimal number of abnormal predicates set to 1 needed to correct the behavior of the circuit, regarding the given counterexamples. The number of fault candidates with cardinality k is reported in column $\#s$. The additional column l in Table 2 gives the length of the counterexamples. The run time of the standard debugging approach [16] is given in column $Standard\ Dbg.$ The next three columns show the run time for our framework: the unsat core processing ($proc$), the percentage of suspects removed from the consideration ($sred.$), and the run time to determine all solutions ($Dbg.$). The last column ($Total\ impr.$) shows the quotient between the total run time to determine all fault candidates of size k using standard debugging versus our framework.

[1]When abnormal predicates were inserted for all gates instead of suspects only, the run time of the second stage increased by a factor up to three.

Table 1: Combinational circuits

	Settings				*Std.*	*Unsat. core diag.*			
circuit	$\#g$	$\#c$	k	$\#s$	*Dbg.* (sec.)	*proc* (sec.)	*sred.* (%)	*Dbg.* (sec.)	*Total impr.*
i7	993	3	2	11	3.73	0.98	96.72	0.44	2.63
i8	1932	3	3	395	157.31	3.11	97.19	27.54	5.13
i9	771	5	2	12	5.27	1.52	89.38	0.93	2.15
k2	630	18	1	2	12.68	5.45	99.11	1.41	1.85
misex3	6249	6	2	684	2195.24	13.61	64.56	795.94	2.71
pair	2848	9	5	4480	M.O.	10.81	95.60	1354.32	[a]> 5.28
rot	1133	2	2	30	5.23	0.74	97.71	0.73	3.56
t481	1631	1	1	15	2.47	0.33	98.85	0.21	4.57
table5	1229	11	3	291	200.85	6.36	69.42	76.72	2.42
too_large	2152	3	2	6	94.31	7.62	75.25	3.42	8.54
x1	725	2	2	273	25.36	0.82	93.94	4.98	4.37
x3	1974	5	4	480	525.84	4.08	95.49	48.02	10.09
x4	959	3	3	225	34.18	1.13	96.10	6.03	4.77

[a]The memory out appeared approximately after two hours run time.

Table 2: Sequential circuits

	Settings					*Std.*	*Unsat. core diag.*			
circuit	$\#g$	$\#c$	l	k	$\#s$	*Dbg.* (sec.)	*proc* (sec.)	*sred.* (%)	*Dbg.* (sec.)	*Total impr.*
b03	195	3	9	2	564	54.30	0.82	20.71	48.94	1.09
b04	821	7	10	1	14	41.81	8.51	79.30	15.38	1.75
b09	197	1	18	1	60	5.12	0.48	48.48	3.38	1.33
b12	1297	4	2	1	15	8.02	1.49	97.23	1.49	2.69
phase-dec	1834	9	10	2	37	583.88	64.46	76.82	210.35	2.12
s1196	818	3	4	1	37	11.13	1.17	93.14	2.76	2.83
s1238	833	14	3	1	2	4.44	2.93	97.64	0.87	1.17

For combinational circuits the number of suspects was reduced by up to 99% (see Table 1). This led to a speed-up of up to 10 times. The extraction of unsat cores is quite efficient. For instance, for $misex3$ the 2 cores were obtained by the SAT solver using propagation only, in less than 1% of the run time of the standard debugging method. The cores are disjoint, therefore small sets of suspects with an exact limit of 1 are created. Debugging directly starts with fewer suspects in separately constrained sets.

For small fault candidates ($k \leq 2$) and circuits with less than 1000 abnormal predicates, the standard debugging approach often needs less than one second. In such a case unsat core extraction causes an overhead. In contrast, for the more interesting case of multiple faults with $k > 2$, the run time is significantly reduced by using our framework. For instance, for $x3$ all fault sites were extracted within one minute, whereas the standard debugging approach needs more than eight minutes.

The results in Table 2 show that the sequential circuits were corrected quite easily; k ranges between 1 and 2 only. Often, modifying a few gates is sufficient to control the faulty outputs. In most sequential cases, the SAT solver finds the suspects by using fast propagation only. For instance, it takes around 1 second for the circuit $s1196$ to reduce the number of suspects to less than 7%. Due to this reduction speed-ups of up to 2.83 of the total diagnosis run time were achieved.

So far the experiments yielded the highest improvements for combinational circuits with $\#g > 1000$ and $k \geq 2$. But for $k \geq 5$ the enumeration of *all* candidate fault sites starts to be the limiting

Table 3: Large fault candidates

Settings					*Std. Dbg.*	*Unsat. core diag.*	
circuit	$\#g$	$\#c$	$\#f$	k	(sec.)	*proc* (sec.)	*inters.* (sec.)
pair	2848	9	5	5	615.58	10.81	10.79
pair	2848	25	10	6	7773.01	38.66	40.47
pair	2848	38	15	7	M.O.	63.38	63.33
pair	2848	19	20	10	T.O.	43.94	42.68
pair	2848	92	25	14	M.O.	3924.90	1198.38

factor. Moreover, in practice it is questionable whether one is interested in a large set consisting of fault candidates that contain many gates. Here, finding a few solutions or even a single fault candidate is more important. Therefore, only one fault candidate is calculated for multiple faults with $k \geq 5$ in the following. The results in Table 3 compare the standard debugging method (column $Std.\ Dbg$), the unsat core diagnosis (Section 4 + Equation (3), column $proc$), and the improvement when intersections of unsat cores are considered as discussed in Section 3.3 (column $inters.$). All algorithms were configured to stop after finding the first fault candidate.

In Table 3 the results for the combinational circuit $pair$ with up to 25 injected faults (column $\#f$) are given. These faults were repaired by a fault candidate of cardinality k. The standard debugging algorithm needs by far more run time than the other two. Often the memory out (M.O.) or time out (T.O.) was reached, even before a first solution was found. In comparison, the unsat core diagnosis finds the first solution with significantly less effort.

The run time of our framework with and without considering intersections is similar for most testcases, except the last one where k is large. In case of $k = 14$, even our framework stops after one hour when intersections of cores are not taken into account. A detailed analysis showed, that the first 13 unsat cores are non intersecting and the 14th core intersects some of them. This initially led to limit the 429 suspects by $k = 14$. This limitation theoretically leave $\binom{429}{14}$ possible fault candidates which does not efficiently support the search process. But the tighter limitations enforced for intersections of cores solve this issue. The first solution was found within 1200 seconds.

In summary, the suspects and as a result the debugging time are reduced by using unsat cores in presence of multiple faults. Especially, in case of fault candidates with large cardinality additional optimizations are useful.

6. CONCLUSIONS

The proposed debugging framework exploits the knowledge of unsatisfiable cores. The solutions were proven to be identical to that of standard SAT-based debugging. Especially for the computationally hard case of multiple faults the run time was significantly reduced.

It remains future work to consider the sequential case more thoroughly, for instance by considering the distance in time between a fault candidate and the observation of a fault [7]. Moreover, exploiting incremental SAT within our framework more efficiently by keeping detailed information about learned clauses [14] may further improve the performance.

7. ACKNOWLEDGMENT

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project Herkules under contract no. 01 M 3082.

8. REFERENCES

[1] K. Chang, I. Markov, and V. Bertacco. Fixing design errors with counterexamples and resynthesis. In *ASP Design Automation Conf.*, pages 944–949, 2007.

[2] N. Eén and N. Sörensson. An extensible SAT solver. In *SAT 2003*, volume 2919 of *LNCS*, pages 502–518, 2004.

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

[4] E. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Design, Automation and Test in Europe*, pages 886–891, 2003.

[5] D. W. Hoffmann and T. Kropf. Efficient design error correction of digital circuits. In *Int'l Conf. on Comp. Design*, pages 465–472, 2000.

[6] J. Huang. MUP: A minimal unsatisfiability prover. In *ASP Design Automation Conf.*, pages 432–437, 2005.

[7] S.-Y. Huang. A fading algorithm for sequential fault diagnosis. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*, pages 139–147, 2004.

[8] M. Liffiton and K. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. In *Journal of Automated Reasoning*, volume Online First. Springer Netherlands, 2007.

[9] C.-C. Lin, K.-C. Chen, S.-C. Chang, M. Marek-Sadowska, and K.-T. Cheng. Logic synthesis for engineering change. In *Design Automation Conf.*, pages 647–651, 1995.

[10] J. Marques-Silva and K. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. on Comp.*, 48(5):506–521, 1999.

[11] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conf.*, pages 530–535, 2001.

[12] Y. Oh, M. N. Mneimneh, Z. S. Andraus, K. A. Sakallah, and I. L. Markov. AMUSE: a minimally-unsatisfiable subformula extractor. In *Design Automation Conf.*, pages 518–523, 2004.

[13] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95, 1987.

[14] O. Shacham and K. Yorav. On-the-fly resolve trace minimization. In *Design Automation Conf.*, pages 594–599, 2007.

[15] O. Shtrichman. Pruning techniques for the SAT-based bounded model checking problem. In *CHARME*, volume 2144 of *LNCS*, pages 58–70, 2001.

[16] A. Smith, A. Veneris, M. Fahim Ali, and A.Viglas. Fault diagnosis and logic debugging using boolean satisfiability. *IEEE Trans. on CAD*, 24(10):1606–1621, 2005.

[17] S.Safarpour, M.Liffton, H.Mangassarian, A.Veneris, and K.A.Sakallah. Improved design debugging using maximum satisfiability. In *Int'l Conf. on Formal Methods in CAD*, 2007.

[18] S. Staber, G. Fey, R. Bloem, and R. Drechsler. Automatic fault localization for property checking. In *Haifa Verification Conference*, volume 4383 of *LNCS*, pages 50–64. Springer, 2006.

[19] A. Veneris and I. N. Hajj. Design error diagnosis and correction via test vector simulation. *IEEE Trans. on CAD*, 18(12):1803–1816, 1999.

[20] J. Whittemore, J. Kim, and K. Sakallah. SATIRE: A new incremental satisfiability engine. In *Design Automation Conf.*, pages 542–545, 2001.

[21] L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Design, Automation and Test in Europe*, pages 880–885, 2003.