Experimental Studies on SMT-based Debugging

Andre Sülflow Görschwin Fey Rolf Drechsler

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

Abstract

SAT-based debugging is a method to automate the debugging process that works quite well on the Boolean level. But on circuits with large arithmetic structures the underlying SAT solver – a Boolean proof engine – often does not finish within the required resource limits. Thus, new solving techniques are required to overcome the gap. Solvers for Satisfiability Modulo Theory *(SMT) provide a higher level of abstraction by combining SAT with theory solvers on the word level. This allows compact handling of many hardware components.*

In this work we focus on debugging with SMT, i.e. SMT-based debugging. *An evaluation on combinational and sequential models on RTL is given. For more than 90% of the instances our experimental studies show significant run time improvements of SMT-based debugging over SAT-based debugging.*

Keywords: Debugging, Boolean Satisfiability (SAT), Satisfiability Modulo Theory (SMT)

1 Introduction

The complexity of circuit designs increases rapidly and requires efficient automation tools throughout the design process. Verification is one of today's major bottlenecks. Efficient methods based on e.g. simulation or formal verification can show faulty behavior, but the detection of fault candidates is often still a manual, time consuming process.

Semi-automatic debugging methods have been developed and rely e.g. on simulation or on structural similarities in combination with manual post-processing. An initial automatic approach based on *Boolean Satisfiability* (SAT) was proposed in [25]. Given a failure trace, the approach automatically returns a set of fault candidates, i.e. locations where the actual fault may be corrected. Additionally, SAT-based debugging provides assignments for the fault candidates that resolve the conflict. This can be used for automatic correction [8].

Today, SAT-based debugging approaches usually rely on a gate level representation. That is, a *Register Transfer Level* (RTL) description of a circuit is translated into a gate level representation, extra logic is added for debugging and the problem instance is given to a SAT solver to obtain a solution. Especially for circuits with many arithmetic components, like e.g. multipliers, the complexity for SAT solving increases. To overcome this, the basic SAT-based debugging approach has been improved by considering e.g. hierarchical information [15, 16], abstraction [24], maximum satisfiability [23] and proofs of unsatisfiability [26]. But in all these cases the SAT solver as a Boolean reasoning engine remains the bottleneck. Run time remains a crucial issue. Thus, there is a need for improvements to handle large circuits.

In recent years solvers for *Satisfiability Modulo Theory* (SMT) were developed and very successfully applied for formal verification of software. The application of SMT to software verification problems leads to smaller run times in particular for large models [1, 2]. SMT solvers internally handle more abstract constraints, e.g. multiplication can directly be represented without transforming the integer problem to the Boolean level. Internally, SMT solvers provide a mixture of theory solving in combination with SAT solving. A SAT solver obtains a solution for an abstract formula, that is checked with a theory solver. Different theories can be instantiated and in particular the bit-vector theory is suitable for modeling hardware. By this, additional high level information of a model is available within the solving process. Using these improvements for hardware at the RTL is very promising.

General studies on the application of SMT to formal hardware verification problems show the potential as future solving technique [27]. In this work an evaluation of SMT in the context of debugging on RTL is given, i.e. *SMT-based debugging*.

In experimental studies combinational and sequential RTL models are considered. For the benchmarks SMT-based debugging outperforms SAT-based debugging in over 90% of the instances.

The paper is structured as follows: In Section 2 the solving techniques SAT and SMT are introduced. An overview of SAT-based debugging is given in Section 3. Section 4 describes and discusses the model for SMT-based debugging. The experimental studies are presented in Section 5. Finally, Section 6 concludes the paper.

2 Proof Engines

The recent advantages in SAT solving led to new methods and tools for formal hardware verification. But, Boolean SAT solving reaches its limit for verification of circuits with large arithmetic structures. Arithmetic circuits are known to be a hard problem for pure Boolean SAT and the verification run time increases significantly. Therefore there is a need for new solving techniques on higher levels of abstraction. At the same time a fully automatic proof without user interaction is required – which is typically not true for theorem provers.

In the following we give an overview of today's state-of-the-art solving techniques SAT and SMT and compare both techniques in the context of formal verification on RTL.

2.1 Boolean Satisfiability

In the last years several improvements in *Boolean Satisfiability* (SAT) have been developed and led to strong SAT solvers [21, 13]. Thus, today SAT solvers are widely applied in formal hardware verification.

The *Boolean Satisfiability problem* (SAT problem) is to decide whether an assignment for a Boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$ exists, so that f evaluates to 1. If there exists such an assignment, the instance is called *satisfiable*, otherwise *unsatisfiable*.

Most SAT solvers rely on the DPLL algorithm [9] extended by conflict analysis [21]: (1) assign values to free variables, (2) determine and propagate implications and (3) analyze conflicts and backtrack.

Normally, the input function f is given in *Conjunctive Normal Form* (CNF). Therefore, each component of a circuit, e.g. gates (AND, OR) or arithmetic components (ADD, MUL), is translated to CNF. A CNF contains clauses and each clause consists of literals where a literal is a variable or its negation. A CNF is satisfied if there exists an assignment for the variables that satisfies each clause. A clause is satisfied if at least one literal is 1. The SAT solver has to find a satisfying assignment or to provide a proof that no such assignment exists.

2.2 Satisfiability Modulo Theory

Satisfiability Modulo Theory (SMT) can be seen as SAT solving on a higher level of abstraction. The satisfiability problem is defined on a set of variables and constraints are given on word level instead of Boolean level.

Variables may be a Boolean predicate, a variable with a fixed bit-width or an uninterpreted function. Among others types, Boolean constraints (e.g. AND, OR), arithmetic constraints (ADD, MUL), as well as relations (e.g. <, >) are supported. Together with a set of assumptions and formula(s) the problem is to find an assignment for the variables, that fulfills all constraints.

The general SMT algorithm is a combination of the enhanced SAT DPLL algorithm [21] with a theory solver resulting in DPLL(T) [19, 11]. In general, (1) a SAT solver is called to get a (partial) non-conflicting assignment on the Boolean abstraction, followed by (2) checking the consistency of the theory constraints using a theory solver. If the theory solver determines a conflict, the theory solver propagates the conflict to the SAT engine. Otherwise the assignment is legal and a satisfying assignment was found. Often an SMT solver provides a proof for the unsatisfiability, if no satisfying assignment exists.

The common SMT input format [22, 3] defines several theories, e.g. for linear arithmetic (QF_LIA), bit-vector (QF_BV) or arrays in combination with bit-vectors (QF_AUFBV). In this work we focus on the most suitable theory for circuit elements: Quantifier free bit-vector theory (QF_BV). QF_BV provides support for logic operations (e.g. AND, OR, XOR) as well as arithmetic (e.g. ADD, MUL, DIV) and control statements (e.g. ITE). Therefore each element of a combinational RTL circuit directly corresponds to a primitive in QF_BV.

Today, the state-of-the-art SMT solvers for QF_BV [6, 10, 18, 5, 12] benefit from the given high level information in the model. That is, a preprocessing step simplifies the model by using term rewriting or abstraction. If this leads to a conflict, the instance is proven unsatisfiable. Otherwise "bit-blasting" is applied. The model is passed to a standard SAT solver (e.g. [13]) to determine the simplified instance satisfiable or unsatisfiable.

3 Debugging with SAT

In the following the basics of SAT-based debugging on combinational and sequential circuits are presented. The debugging approaches use a faulty circuit and one or more counterexamples (failure traces) as input. Each counterexample contains additional information on the expected behavior, e.g. the expected values on the primary outputs or a failing property that has to be fulfilled. For more details on the debugging algorithm we refer to [25] and for property debugging to [17].

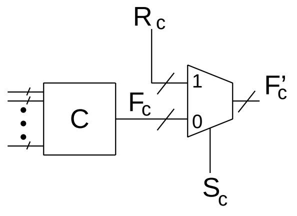


Figure 1. Correction logic

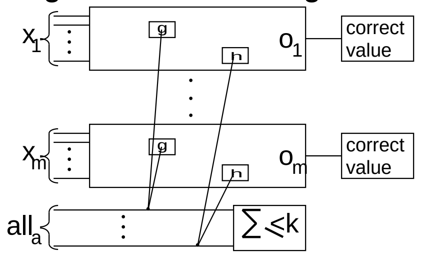


Figure 2. Combinational Debugging

First, the circuit is divided into components. Components are possible fault candidates and the choice controls the granularity of debugging. Typical choices are gates or expressions, but also hierarchical or structural information are taken into account [15, 16]. For each component additional *correction logic* is inserted to change the output behavior of a component (see Section 3.1). Afterwards the debug SAT-instance is created for a given a set of failure traces (see Section 3.2) or a property and a counterexample (see Section 3.3).

3.1 Correction Logic

To automatically debug the cause of faulty behavior, for each component extra correction logic is inserted that allows the component to behave non-deterministically. The original function F_c of a component C is replaced by F_c' as shown in Figure 1. The select line of the multiplexer controls F_c': if S_c is zero, then $F_c' = F_c$, otherwise $F_c' = R_c$. S_c is also called *abnormal predicate*. R_c is a free variable and can have any value, in particular, R_c may be a Boolean as well as a multi-bit signal.

3.2 Combinational Circuit Debugging

The approach requires a model of the circuit, a set of failure traces and a set of correct output responses. This information is provided e.g. by formal equivalence checking or by a failing simulation trace from a test bench.

Given a combinational circuit C, a set of failure traces $X_1, \ldots, X_m$ and a set of corresponding correct output responses $O_1, \ldots, O_m$, the SAT instance is created as in Figure 2 [25]: For each failure trace the circuit is replicated, the correction logic is added, the failure trace is applied on the input signals and the correct output values are constrained. The same abnormal predicate is used for a component over all replicated instances. That is, if an abnormal predicate of a component is activated, the component behaves non-deterministic in all replicated instances simultaneously.

The total number of activated abnormal predicates is limited to k. Limiting k to zero, i.e. no correction is allowed on any component, leads to an unsatisfiable instance due to forcing correct output responses. While the SAT instance is unsatisfiable, k is incremented by 1. If the activation of k abnormal predicates leads to a satisfying assignment, then all fault candidates are extracted. For each satisfying assignment, i.e. each fault candidate, a blocking clause is added to the SAT instance. The algorithm terminates if the instance becomes unsatisfiable and each fault candidate with minimal cardinality k is extracted.

The extension to the sequential problem is straightforward: the circuit is unrolled and the same abnormal predicate is used for a component in all time steps and for all replicated instances.

Several techniques to improve the efficiency of the basic approach have been proposed (see Section1), but the underlying model essentially remains the same.

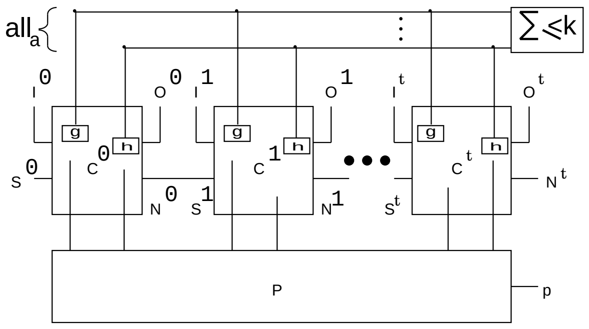


Figure 3. Property Debugging

3.3 Property Debugging

If property checking [4] fails, not only a counterexample is provided, but also the specification for the correct output response is given: the property. Instead of constraining each output directly to correct values, the property describes the desired relation between a set of signals.

The problem instance for debugging properties is a mix of the one for SAT-based bounded model checking [4] and sequential SAT-based debugging. From a given faulty circuit C, a property P of length t and a counterexample the debug instance is created (see Figure 3) [17]. The circuit is unrolled for t time steps and the state variables $S^1, \ldots, S^t$ are connected to the corresponding state variables $N^0, \dots, N^{t-1}$. For each component the correction logic is added and the same abnormal predicate is used for a component at all time steps. Afterwards the property is connected to the unrolled instance.

A given counterexample contains primary inputs $I^0, \dots, I^t$ and the initial state S^0. For the debugging process, the values of the primary inputs and initial states are constrained on the SAT instance. By forcing the output p to fulfill the property and k to zero the SAT instance becomes unsatisfiable.

As in combinational debugging the algorithm incrementally increases k and computes the fault candidates with minimal cardinality.

4 Debugging with SMT

This section focuses on the application of SMT solvers to the debugging problem. A presentation of the debugging instance for SMT is given and possible benefits of the higher level of abstraction are highlighted.

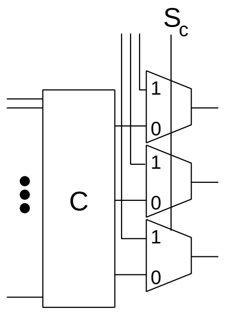
In general, the model of Section 3 is adopted for SMT-based debugging without modifications: A debug instance is created by translating the circuit to SMT, adding extra correction logic and limiting the number of abnormal predicates to k. Afterwards an algorithm increments k until a satisfiable solution is obtained and the fault candidates are extracted. But, as shown in the following, debugging on SMT level has additional advantages over SAT.

4.1 Debugging Instance

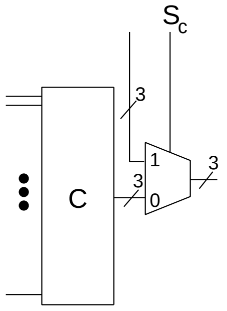
Without loss of information, all RTL constructs of a debugging instance are translatable one-to-one to the SMT input format [22, 3]. Obviously, the number of SMT constraints is far less than the number of clauses in SAT [27]. That is, a multiplier of width n is compactly represented by one constraint (*bvmul*), two n-bit input variables and one n-bit output.

The representation of the correction logic in SMT is more compact, too. In Figure 4 the correction logic for a component with an output of bit-width three is shown. Instead of creating three single-bit multiplexers as in SAT, only one constraint is required for a component with a three-bit output. Similar to hierarchical debugging the outputs of a component are combined to one output [15, 16]. The hierarchical information is naturally given by the design and no extra structural analysis has to be done. Like in SAT-based debugging more complex modules, like e.g. an encoder or decoder, in the system may also be used as components in SMT debugging.

The efficient formulation of cardinality constraints in SAT has been studied intensively, e.g. recently in [14]. Different ways to transform these constraints may lead to drastically different sizes of the resulting CNF representation.



(a) SAT



(b) SMT

Figure 4. Multiple bit correction logic for SAT and SMT

Moreover, the transformation has a side-effect on the performance of the implication engine. Some of these arguments can directly be transferred to the SMT domain. In our application, the limitation constraint structure is taken from a *Binary Decision Diagram* (BDD) [7] that is mapped to Boolean multiplexer constraints. That is, the overhead in terms of constraints and clauses is similar in SAT and SMT. Alternative approaches use e.g. adders [14]. But, for a large number of abnormal predicates many adders with a large bit-width are required. Due to this overhead, this representation is not considered here.

4.2 Discussion

SAT was shown to be very powerful for circuits given on pure Boolean level. But the verification of arithmetic operations is a known hard problem. Therefore, if the RTL description contains arithmetic elements, they are translated first to gate level and then to CNF. But, the transformation destroys higher level information on the functionality and the circuit structure. This often makes the formal verification with a SAT solver difficult.

SMT solvers use the state-of-the-art SAT techniques and combine them with high level optimization. Instead of passing the whole circuit to a SAT solver the instance is first simplified, so that the remaining instance is in most cases smaller and easier to solve. As shown in the last SMT competition, especially the simplification process is one of the key techniques to solve verification problems on word level – the four highest ranked solvers in QF_BV (bit vector arithmetic) use simplification techniques before running a SAT solver [2].

As shown in Section 4.1 the representation of the debugging problem is more compact for SMT. The compact representation in combination with efficient preprocessing algorithms may lead e.g. to a lower memory consumption and smaller run times for an SMT solver. Considering these facts, the simplification techniques applied in SMT solvers induce some overhead at first. Therefore simple instances and those where simplification is not possible should directly be transformed into a SAT problem to save run time. But whenever the problem instance is hard (e.g. only a few solutions in a large search space) and high level simplification techniques are applicable, the SMT solver should be faster. Unfortunately, it is not predictable which instances are simple and which ones are hard. For the benchmarks considered in the experiments, the number of constraints is a good indicator – where small instances are simple. Of course, in general also a small instance may be very hard to solve.

At this stage the problem has been directly transformed into an SMT problem. But SMT provides more powerful constraints than SAT. In particular uninterpreted functions [3] may be a powerful instrument for debugging. The simple SAT based formulation allows components to behave non-deterministically (because primary inputs are used). But in a circuit such non-deterministic behavior is not possible (or not wanted). As a result SAT-based debugging may return fault candidates where a correction is not possible [17] and elaborate techniques are required when considering correction [8]. Exploiting such additional features of SMT to improve the debugging effectivity is future work.

5 Experimental results

In an experimental study the SMT-based debugging approach is applied and evaluated on combinational and sequential models given on RTL.

The combinational models are obtained from benchmarks used at the last SMT competition [2]. Because the SMT instances do not explicitly model primary inputs and outputs, we modified the instance as follows: (1) Each named variable is considered as a primary input and (2) all assumptions and the formula are not restricted to be true, but are modeled as primary outputs. Therefore, the number of free variables is increased and the search space becomes larger. All together 80 combinational instances with 200 to 40000 Boolean and bit-vector operations are considered.

For property debugging a sequential system from the railway domain is considered [20]. The properties argue over up to three time frames.

For all instances a fault is injected by changing an operation, randomly. That is, e.g. an *EQUAL* constraint is replaced by *LESS* or an *AND* with an *OR*. Afterwards property checking or equivalence checking was applied to generate one counterexample for the debugging process. Each operation is taken as a separate component in SAT as well as in SMT. In particular, the same components are used for SAT- and SMT-based debugging.

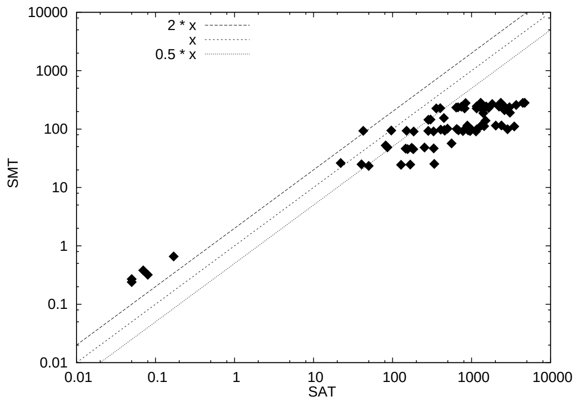


Figure 5. Combinational Debugging

SAT-based debugging is evaluated on the base of MiniSat [13]. For SMT-based debugging we considered STP [18] as core engine[1]. The debugging process computed all possible candidate fault sites with minimal cardinality. The experiments were carried out on a Dual-Core AMD Opteron 2220 SE with 32 GB of main memory.

The run times given in CPU seconds for debugging of combinational models are presented in Figure 5. A logarithmic scale is used on both axes. The five instances in the lower left consist of 200 to 250 operations only. Here, SAT is faster than SMT, but the overall run time is small. The reason may be the more complex parser and pre-processing process for the SMT instances.

In over 90% of the instances SMT outperforms SAT-based debugging significantly. These are all instances in the upper right corner with more than 10000 operations. Especially if a large number of arithmetic operations are taken into account, the performance of SAT degrades. In total a run time improvement of 10 is observed for SMT in comparison to SAT.

Similar observations are made in case of property debugging. Table 1 shows the number of fault candidates ($\#sol$), the run times for SAT and SMT and the improvement of SMT in comparison to SAT (improv.). The model consists of 557 components. SMT outperforms SAT in all cases.

[1]We have also evaluated Boolector [6] as winner of the last SMT competition, but for our benchmarks the run time was 5 to 10 times slower than STP, which may be due to the heuristics involved in the search process.

Table 1. Property Debugging

Property	$\#sol$	SAT	SMT	improv.
state_equal	37	76.09	21.35	3.56
state_0	17	23.73	7.52	3.16
state_1	10	31.35	4.59	6.83
state_2	14	47.45	6.31	7.52
state_3	15	42.87	6.70	6.40

6 Conclusion

In this work SMT-based debugging was evaluated for debugging on RTL. In experimental studies SMT-based debugging outperformed SAT-based debugging for circuits with a large number of arithmetic structures. Run time improvements of up to a factor of 10 were achieved. For future work we focus to exploit more sophisticated SMT techniques, like, e.g. uninterpreted functions.

References

[1] C. Barrett, L. de Moura, and A. Stump. SMT-COMP: Satisfiability modulo theories competition. In K. Etessami and S. Rajamani, editors, *17th International Conference on Computer Aided Verification*, pages 20–23. Springer, 2005.

[2] C. Barrett, M. Deters, A.Oliveras, and A. Stump. SMT-COMP: Satisfiability modulo theories competition. `www.smtcomp.org`, 2008.

[3] C. Barrett, S. Ranise, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2008.

[4] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *LNCS*, pages 193–207. Springer Verlag, 1999.

[5] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Schulz, and R. Sebastiani. The MathSAT 3 System. In *Int'l Conference on Automated Deduction (CADE)*, volume 3632, pages 315–321, 2005.

[6] R. Brummayer and A. Biere. Boolector 0.4. In *SMT-COMP: Satisfiability Modulo Theories Competition*, 2008. Available at http://fmv.jku.at/boolector/.

[7] R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Comp.*, 35(8):677–691, 1986.

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

[9] M. Davis, G. Logeman, and D. Loveland. A machine program for theorem proving. *Comm. of the ACM*, 5:394–397, 1962.

[10] L. de Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008. Available at http://research.microsoft.com/projects/Z3.

[11] B. Dutertre and L. Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Computer Aided Verification*, volume 4114, pages 81–94, 2006.

[12] B. Dutertre and L. Moura. The YICES SMT Solver. In *SMT-COMP: Satisfiability Modulo Theories Competition*, 2006. Available at http://yices.csl.sri.com/.

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

[14] N. Eén and N. Sörensson. Translating pseudo-boolean constraints into sat. In *Journal on Satisfiability, Boolean Modeling and Computation*, volume 2, pages 1–26, 2006.

[15] 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.

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

[17] G. Fey, S. Staber, R. Bloem, and R. Drechsler. Automatic fault localization for property checking. *IEEE Trans. on CAD*, 27(6):1138–1149, 2008.

[18] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification*, number 4590 in LNCS, 2007.

[19] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In *Computer Aided Verification*, volume 3114 of *LNCS*, pages 175–188, 2004.

[20] S. Kinder and R. Drechsler. Modeling and formal verification of counting heads for railways. In *Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT)*, pages 64–76, 2008.

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

[22] S. Ranise and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2006.

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

[24] S. Safarpour and A. Veneris. Abstraction and refinement techniques in automated design debugging. In *Design, Automation and Test in Europe*, pages 1182–1187, 2007.

[25] 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.

[26] A. Sülflow, G. Fey, R. Bloem, and R. Drechsler. Using unsatisfiable cores to debug multiple design errors. In *Great Lakes Symposium on (GLSVLSI)*, pages 77–82, 2008.

[27] A. Sülflow, U. Kühne, R. Wille, D. Große, and R. Drechsler. Evaluation of SAT like proof techniques for formal verification of word level circuits. In *IEEE Workshop on RTL and High Level Testing (WRTLT)*, pages 31–36, 2007.