WoLFram – A Word Level Framework for Formal Verification

André Sülflow Ulrich Kühne Görschwin Fey Daniel Große Rolf Drechsler
Institute of Computer Science
University of Bremen, 28359 Bremen, Germany
Email: {suelflow,ulrichk,fey,grosse,drechsle}@informatik.uni-bremen.de

Abstract

Due to high computational costs of formal verification on pure Boolean level, proof techniques on the word level, like Satisfiability Modulo Theories (SMT), were proposed. Verification methods originally based on Boolean satisfiability (SAT) can directly benefit from this progress.

In this work we present the word level framework WoLFram *that enables the development of applications for formal verification of systems independent of the underlying proof technique. The framework is partitioned into an application layer, a core engine and a back-end layer. A wide range of applications is implemented, e.g. equivalence and property checking including algorithms for coverage/property analysis, debugging and robustness checking. The back-end supports Boolean as well as word level techniques, like SMT and Constraint Solving (CSP). This makes* WoLFram *a stable backbone for the development and quick evaluation of emerging verification techniques.*

1. Introduction

Following Moore's law, the number of components that can be integrated on a chip increases exponentially over the years. Already today the verification gap – very large designs can be manufactured, but not verified due to complexity issues – is one of the biggest challenges for the hardware industry. Thus, the need for efficient verification techniques grows.

Formal verification methods based on *Boolean Satisfiability* (SAT) were shown to work efficiently for circuits on the Boolean level. But for circuits with large arithmetic structures, like adders and multipliers, pure Boolean SAT solving is running out of steam. Therefore several new verification techniques on the word level, e.g. *predicate abstraction* and *Satisfiability Modulo Theory* (SMT), have been developed. Word level techniques speed-up the verification process significantly [1], [2]. Some of the verification methods based on Boolean SAT techniques can directly benefit from the advances for the word level provers. There have been several publications on adapting existing methods for the use of advanced proof techniques, like [3], [4], [5], [6], [7].

In this work we present a framework for the generic development of formal verification methods that are independent of the underlying proof technique, called WoLFram (*Word Level Framework*). The framework is partitioned into an application layer, a core engine and a solver back-end. The motivation for this is on the one hand the easy adaptation of existing verification methods for new proof techniques. On the other hand, by building new methods on top of our core data structures, we can reuse both, the parsing front-ends and the solver back-end. In this way, for a new algorithm there are a number of input languages (like *SystemC* or *Verilog*) and numerous state-of-the-art solvers directly available.

The input of WoLFram is a *word level netlist* (wlnetlist) that may contain Boolean as well as word level operations. Synthesis tools for *SystemC*, *Verilog* and *Blif* are available. Integrated applications are ranging from simulation, property checking and equivalence checking to algorithms for analyzing coverage, debugging and computing robustness. For each of these applications, several proof engines are available, in particular Boolean SAT, *Pseudo Boolean SAT* (PBS) as well as word level techniques, like SMT or *Constraint Satisfaction Problem* (CSP).

WoLFram allows for a fair and easy evaluation of proof techniques to find the best suited solving paradigm for newly developed verification methods (see e.g [4]). Even for a single application it may be useful to apply alternating proof techniques on different types of designs.

Overall, WoLFram provides a stable and extensible framework for formal verification and allows the fast development of formal verification methods.

The paper is structured as follows: Related work is discussed in Section 2. Section 3 gives an overview of the framework, followed by a detailed description of the input model (Section 4), implemented applications (Section 5) and proof engines (Section 6). The verification capabilities are presented in a case study for a RISC CPU in Section 7. Finally, a summary is given in Section 8.

2. Related Work

In 2005 *SyCE* – An Integrated Environment for System Design in SystemC – was developed in our group [8]. WoLFram is a generalization of SyCE. While SyCE focuses on *SystemC* designs and Boolean proof techniques, WoLFram abstracts from a specific HDL and supports Boolean as well as word level techniques.

Academic tools for formal verification on the Boolean level or on word level descriptions are e.g. [9], [10], [8],

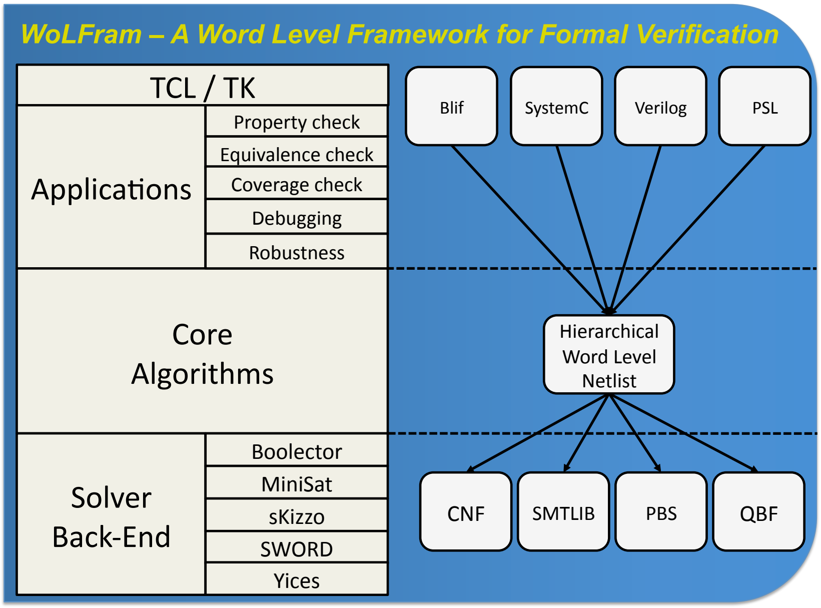


Figure 1. System overview

[11]. But all the tools are either restricted to a specific *Hardware Description Language* (HDL), an application or a proof technique. For example, [10] is a formal verification environment for *C* and [8] for *SystemC* only. On the other hand [9] is specialized on model checking and does not contain support for other applications using formal methods, like debugging or robustness computation. Boolean models and techniques are limiting the approach of [11]. The strong verification techniques behind commercial tools like [12], [13], are not fully known to the public. An approach for the formal verification of software is presented in [14]. There, an intermediate representation is used, enabling the modeling of Boolean and word level operations for software. However, the focus of *WoLFram* is the formal verification of hardware designs.

Overall, the prospective need for verification techniques on a higher level of abstraction requires a flexible framework with a wide range of applications and proof techniques. The integration in *one* framework supports a high degree of reuse and a fast evaluation of new verification techniques.

3. System Overview

The overall flow of *WoLFram* is shown in Figure 1. Here, the left-hand side represents the algorithmic flow and the right-hand side the data flow.

The control flow of *WoLFram* is partitioned into an application layer, a core engine and a back-end layer. The applications are written separately from the core algorithms of *WoLFram*. An API of the core engine provides access to frequently used features. Additionally, each of the applications is configurable with different back-end proof engines. In this way, different proof techniques can be evaluated without changing the application.

The core engine contains classes for the representation of a hierarchical *word level netlist* (wlnetlist), counterexample generation and simulation. The simulation supports the generation of wave traces in VCD format, which can be visualized with standard tools. Common algorithms like path tracing, topological sorting, reachability analysis and design abstraction facilities provide a rich set of basic functionality for the applications.

The supported input languages and verification techniques are presented on the right side of Figure 1. Front-ends for *SystemC* and *Verilog* synthesize a design to a wlnetlist (see Section 4). Afterwards, an application is started and works on the hierarchical wlnetlist. By choosing a proof engine, the core engine automatically translates the formal problem description either to *Conjunctive Normal Form* (CNF), the common input format for Boolean SAT solvers, to *Pseudo Boolean SAT* (PBS) [15], to *Quantified Boolean Formulae* (QBF) [16] or to *SMTLIB* format [17].

WoLFram is implemented in C++ and a set of over 200 unit tests ensures a stable development environment. A TCL layer on top of *WoLFram* connects to a *Graphical User Interface* (GUI), an interactive terminal, and a scripting interface to invoke functions or applications.

The following sections give more details on the input language (Section 4), the applications (Section 5) and the underlying proof techniques (Section 6).

4. Input Language

The input format of *WoLFram* is a description of a hierarchical wlnetlist. The wlnetlist contains information about signals and their connection via operations. Both, the signals as well as the operations, have variable bit size for signed or unsigned values. By this, Boolean gate level descriptions as well as n-bit logical and arithmetic operations are supported. Additionally, arrays of signals and operations enable the compact encoding of memories, register files or similar regular circuit structures.

For example, the following operations for single and multiple bits ($n \geq 1$) are supported:

- Logic and bit operations: e.g. AND, OR, SETBIT
- Arithmetic operations: e.g. ADD, MUL, DIV
- Control and relational operations: e.g. ITE, LEQ, EQ
- Array operations: READ, WRITE

Currently only synchronous circuits with a single clock domain are supported.

The generalization to a wlnetlist separates the input language from a concrete HDL. The signals and operations are the common elements that can be synthesized from any HDL. Additionally, the HDL synthesis tools annotate the wlnetlist with source code information, i.e. line and column number from the original HDL input. A *WoLFram* wlnetlist is stored in *eXtensible Markup Language* (XML) format, simplifying parsing and generation of wlnetlist files.

Three front-ends are implemented to generate the wlnetlist. For example, gate level netlists are synthesizable from Blif and BlifMV format [18]. But also higher level HDLs like *SystemC* and *Verilog* are supported[1]. The *SystemC* front-end is based on the parser from [8].

5. Applications

During the last two years a wide range of applications based on formal methods were implemented and evaluated in the *WoLFram* framework. This includes applications for property and equivalence checking, but also the strong debugging facilities of *WoLFram*. The developed set of applications provides a stable backbone for formal verification and evaluation of new techniques. This section gives an overview of the integrated applications:

- Property checking
- Coverage and property analysis
- Equivalence checking
- Debugging
- Robustness computation

Section 5.1 and Section 5.2 give details on property checking and analysis algorithms, followed by information on equivalence checking in Section 5.3. Debugging and robustness computation are introduced in Section 5.4 and Section 5.5, respectively.

5.1. Property Checking

Functional verification is a major issue in today's hardware design flows. To ensure the correct functionality of a circuit, its specification is formalized in a dedicated verification language such as the *Property Specification Language* (PSL) [19]. The properties are then checked using model checking procedures like *Bounded Model Checking* (BMC) [20].

In *WoLFram* there are two different BMC algorithms implemented, both taking as input safety properties written in the PSL language. The first method, *k-induction*, starts from the initial states and tries to prove the property using induction [21]. The second method, interval property checking, abstracts the initial states, thus performing an all-states verification. While this enables the efficient verification of larger designs, usually additional invariants have to be supplied to prevent spurious counterexamples that start from an unreachable state. Details on this method can be found in [22].

5.2. Coverage and Property Analysis

In order to guarantee a high quality of the functional verification, a coverage check is available. The proof engine is used to decide whether the properties uniquely determine the value of all outputs for each input and state combination. The method implemented in *WoLFram* is described in [23]. If the check fails, a missing scenario (coverage gap) is presented to the user.

Additional methods aid the user in finding a concise description of a design that covers the whole functionality. In [24] an approach to understand the reasons for contradictions in the antecedent of a property has been proposed. As described in [25], properties can be strengthened automatically and in case of a failing coverage check, suggestions are made how to complete the set of properties.

5.3. Equivalence Checking

Equivalence checking decides whether an implementation is functionally equivalent to a reference implementation (specification) or not.

Combinational equivalence is checked by creating a miter circuit [26] from implementation and specification, connecting the primary inputs of both circuits and adding a comparison logic to the primary outputs. If one of the outputs differs for any input combination, implementation and specification are not functionally equivalent and a counterexample is generated.

For sequential circuits an *Iterative Logic Array* (ILA) [27] is created, the primary inputs are connected and the comparison logic is added to the primary outputs. But also name-based state matching algorithms are implemented to avoid unrolling. Algorithms to handle retiming are not implemented at the moment, but are focus of future work.

[1] The synthesis process of HDLs is not in focus of this work and is therefore not considered in detail.

5.4. Debugging

Formal verification techniques like property and equivalence checking show faulty behavior, but often the counterexamples have to be analyzed by a designer in a time consuming manual process.

SAT-based debugging algorithms provide a powerful technique to evaluate the cause for a counterexample automatically. Given a faulty implementation, a set of counterexamples and correct output responses from a specification, a diagnosis application computes a set of components that form a possible cause for the faulty behavior. Components may be gates as well as hierarchical modules of a wlnetlist.

The *WoLFram* framework implements basic SAT-based debugging [28], [29] as well as extensions that use unsat-cores to speed-up the debugging process [30] and increase the accuracy [31]. Experimental studies on SMT-based debugging show promising results [6].

5.5. Robustness Computation

While facing continuously shrinking feature sizes, permanent faults from manufacturing or externally triggered transient faults may affect the correct functionality of circuits. Therefore, the demand for fault tolerance increases.

To ensure robustness, often redundancy is applied during design [32] or at the layout level [33]. But an additional formal check to prove robustness of an implementation is usually not performed.

A formal method to prove fault tolerance was proposed in [34]. Without assuming a specific fault type, the method formally checks, if non-deterministic behavior of a signal is observable at the primary outputs. In this case the signal is non-robust. Recently, the basic method has been extended to the computation of bounds for fault tolerance [35]. Both methods are available in the *WoLFram* environment.

6. Proof Engines

The back-end of *WoLFram* consists of proof engines on the Boolean level and on the word level. Table 1 summarizes the different paradigms and lists the engines available within *WoLFram*. The first column shows the type of proof algorithm. The second column briefly explains the specific properties of the algorithm. Column ‘engine‘ denotes the engines that are currently integrated in *WoLFram*.

A common interface controls the generation of verification models in the respective level. That is, a base class for all verification models defines a generic interface. The interface contains methods for the creation of e.g. logical operations like OR and arithmetic operations like multiplication and addition. Each proof technique inherits from the base class and implements a specific verification model. To evaluate a new proof technique, only a newly derived verification model is required. The translation of circuit related problems considered in *WoLFram* into the respective formats for different solvers takes linear time, but trade-offs of different encodings have to be considered carefully [6], [4].

```
property INCREMENT =
always(
  reset == 0 &&
  pc.le == 0 &&
  pc.pc < 2047
) -> (
  next[1](
   (prev[1](pc.en) == 1) ?
     (pc.pcout == prev[1](pc.pc) + 1) :
     (pc.pcout == prev[1](pc.pc))
  )
);
```

Figure 2. Failing property

For Boolean engines the problem has to be flattened, typically into a two-level representation. In contrast, the word level solvers have access to structural and semantic information about the original problem. For example arithmetic operations are encoded into two-level logic for SAT, but are directly available in SMTLIB format. Therefore techniques like predicate abstraction [49] or term-rewriting can be applied within the word level engine.

7. Case Study

To show the application of *WoLFram* on a concrete design, a case study for a RISC CPU is presented in this section. The case study consists of three parts: visualization (Section 7.1), debugging (Section 7.2) and coverage analysis (Section 7.3).

7.1. Visualization

The visualization engine of Concept Engineering[2] provides a graphical view of the RISC CPU (see Figure 4). The visualization capabilities improve design and fault understanding and may significantly speed-up manual engineering tasks such as debugging and optimization.

Some features are highlighted here:

- navigation in a hierarchical schematic view
- cone extraction
- source code browsing
- cross-probing schematic view and source code

The visualization engine interacts with *WoLFram* over a TCP/IP connection and provides an additional GUI that can be used in an interactive session. Formal applications can use the visualization to annotate counterexamples or to mark important components.

7.2. Debugging

Debugging is applied to find candidate fault sites for a failing property. The property is specified in PSL (see

[2] http://www.concept.de

Table 1. Proof algorithms and engines

Algorithm	Description	Engine
Boolean level		
SAT	*Boolean Satisfiability* finds an assignment for a Boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$ so that f evaluates to 1 or proves that no such assignment exists; common input format is *Conjunctive Normal Form* (CNF)	MiniSAT [36], zChaff [37], BooleForce [38], PicoSAT [39]
PBS	*Pseudo Boolean SAT* also known as 0-1 integer linear programming; extends SAT by constraints of the form $c_1x_1 + c_2x_2 + \ldots + c_nx_n \leq d_0$ where the x_i are Boolean variables, the c_i and d are integer constants	Pueblo [40], MiniSAT+ [15]
QBF	*Quantified Boolean Formulae* extend Boolean SAT by existential and universal quantification; provides speed-up, e.g. for debugging [41]	sKizzo [16], Quantor [42]
Word level		
SMT	*Satisfiability Modulo Theories* integrates SAT engines with theory solvers, e.g. for bit-vector arithmetic; SMTLIB is the common input format [17]	Boolector [43], CVC3 [44], SWORD [45], Yices [46]
CSP	*Constraint Satisfaction Problems*; developed and applied in the domain of artificial intelligence; common input format [47]	Abscon [48]

Figure 2) and checks whether the *program counter* is incremented correctly[3]. In this example the *program counter* is faulty due to swapping the *then* and the *else* part of the *if*-statement in lines 23 to 31 shown at the bottom of Figure 4. The faulty behavior is found by a property check. Now, a counterexample is extracted and annotated in the design. The cone view is used to restrict the view to the *program counter*. In parallel, the corresponding *Verilog* source code is opened at the bottom.

Afterwards, the debugging application determines all fault candidates that may be a fix for the design. Debugging correctly determines the reason for the faulty behavior in the *program counter* and marks the fault candidates red. By selecting a fault candidate in the cone view, in parallel the corresponding part in the source view is marked red. As shown in Figure 4, the above mentioned *if-then-else* block is one of the fault candidates.

7.3. Coverage Analysis

After the design has been fixed, the property holds. Additional properties describe the reset behavior and the loading of new addresses into the program counter (not shown here). Now, an automatic check determines whether the set of properties completely covers the functionality of the examined block. For this purpose, the coverage analysis is started for signal `pcout`, resulting in an uncovered scenario. The corresponding wave trace is shown in Figure 3. As can be seen in the figure, the wrap around behavior of the program counter – restarting with the value zero after exceeding the highest address – has been omitted in the properties. After including a proper description of this case, the coverage check succeeds and thus the property set forms a complete specification.

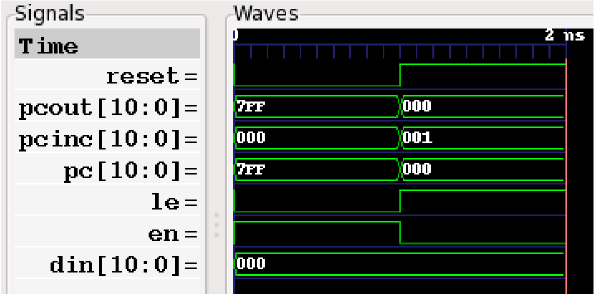


Figure 3. Coverage analysis

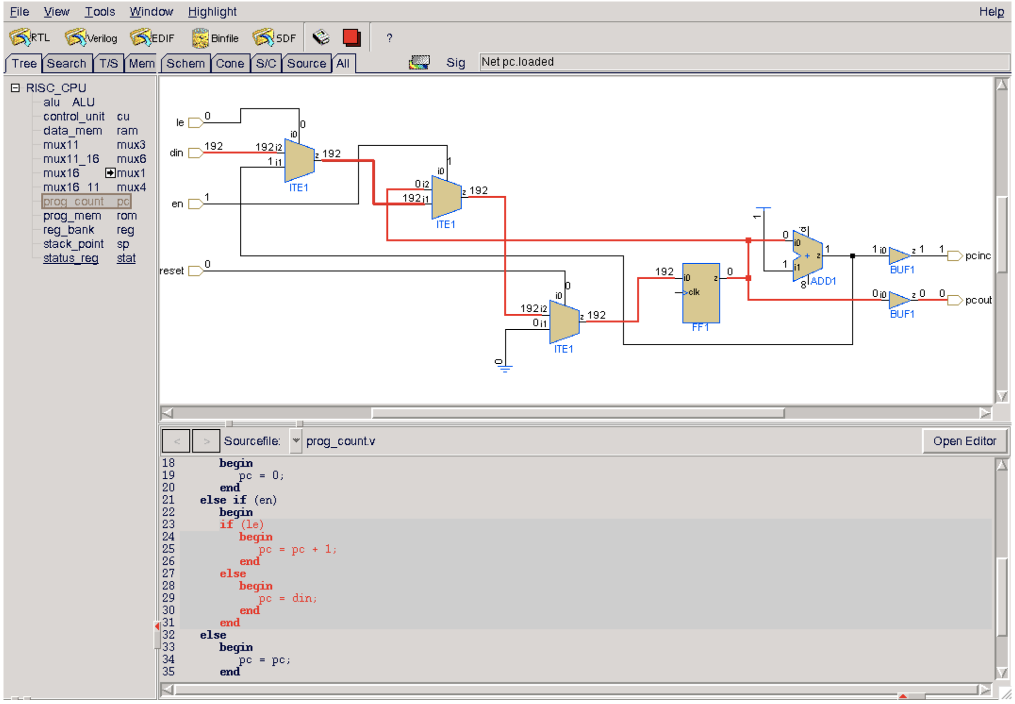
8. Summary

In summary, *WoLFram* is an environment for the development of applications for formal verification on the Boolean level and on the word level. Front-ends for synthesizing HDLs are available and a wide range of applications provide strong verification capabilities. The replaceable back-end allows an easy and fair evaluation of applications using different proof techniques. Several publications clearly demonstrate the strengths of *WoLFram* in different application domains.

Acknowledgment

The techniques described here have been developed in the research projects Herkules (contract no. 01 M 3082) and VerisoftXT (contract no. 01 IS 07008 C) funded by the *German Federal Ministry for Education and Research* (BMBF). We would like to thank Gerhard Angst and Lothar Linhard from Concept Engineering GmbH, Freiburg, Germany for their support. Furthermore, we would like to thank Olaf von der Ahe, Cécile Braunstein, Stefan Frehse, Christian Genz, Finn Haedicke, Marc Messing and Robert Wille for their help during implementation and integration of *WoLFram* and for many helpful discussions.

[3] The prefix “pc.” of a signal name denotes the program counter in the hierarchical CPU model, whereas “pc” is the register of the program counter and en (le) denotes the *enable* (*load enable*) signal.



RISC_CPU	
alu	ALU
control_unit	cu
data_mem	ram
mux11	mux3
mux11_16	mux6
mux16	mux1
mux16_11	mux4
prog_count	pc
prog_mem	rom
reg_bank	reg
stack_point	sp
status_reg	stat

```
      begin
        pc = 0;
      end
    else if (en)
      begin
      if (le)
        begin
          pc = pc + 1;
        end
      else
        begin
          pc = din;
        end
      end
    else
      begin
        pc = pc;
      end
```

Figure 4. Visualization

References

[1] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer Aided Verification*, ser. LNCS, vol. 1855, 2000, pp. 154–169.

[2] A. Gupta and O. Strichman, "Abstraction refinement for bounded model checking," in *Computer Aided Verification*, ser. LNCS, vol. 3576, 2005, pp. 112–124.

[3] M. Ganai and A. Gupta, "Accelerating high-level bounded model checking," in *Int'l Conf. on CAD*, 2006, pp. 794–801.

[4] 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*, 2007, pp. 31–36.

[5] P. Bjesse, "A practical approach to word level model checking of industrial netlists," in *Computer Aided Verification*, ser. LNCS, vol. 5213, 2008, pp. 446–458.

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

[7] A. Armando, J. Mantovani, and L. Platania, "Bounded model checking of software using SMT solvers instead of SAT solvers," *International Journal on Software Tools for Technology Transfer (STTT)*, Feb. 2009.

[8] R. Drechsler, G. Fey, C. Genz, and D. Große, "SyCE: An integrated environment for system design in SystemC," in *IEEE International Workshop on Rapid System Prototyping*, 2005, pp. 258–260.

[9] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking," in *Computer Aided Verification*, ser. LNCS, vol. 2404, 2002, pp. 241–268.

[10] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 2988, 2004, pp. 168–176.

[11] The VIS Group, "VIS: A system for verification and synthesis," in *Computer Aided Verification*, ser. LNCS, vol. 1102. Springer Verlag, 1996, pp. 428–432.

[12] OneSpin Solutions GmbH, *OneSpin Verification Solutions*, http://www.onespin-solutions.com, 2008.

[13] Jasper Design Automation, *JasperGold*, http://www.jasper-da.com, 2008.

[14] P. Farail, P. Gaufillet, F. Peres, J.-P. Bodeveix, M. Filali, B. Berthomieu, S. Rodrigo, F. Vernadat, and H. Garavel, "Fiacre: an intermediate language for model verification in the topcased environment," in *European Congress on Embedded Real-Time Software (ERTS)*, 2008.

[15] N. Eén and N. Sörensson, "Translating pseudo-Boolean constraints into SAT," in *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 2, 2006, pp. 1–26.

[16] M. Benedetti, "sKizzo: A suite to evaluate and certify QBFs," in *Proc. of International Conference on Automated Deduction*, 2005, pp. 369–376.

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

[18] *OCTTOOLS-5.2 Part II Reference Manual*, Electronics Research Laboratory, University of California at Berkeley, Mar. 1993.

[19] *Accellera Property Specification Language Reference Manual, version 1.1*, http://www.pslsugar.org, 2005.

[20] 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*, ser. LNCS, vol. 1579. Springer Verlag, 1999, pp. 193–207.

[21] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a SAT-solver," in *Int'l Conf. on Formal Methods in CAD*, ser. LNCS, vol. 1954. Springer, 2000, pp. 108–125.

[22] M. Nguyen, M. Thalmaier, M. Wedler, J. Bormann, D. Stoffel, and W. Kunz, "Unbounded protocol compliance verification using interval property checking with invariants," *IEEE Trans. on CAD*, vol. 27, no. 11, pp. 2068–2082, Nov. 2008.

[23] D. Große, U. Kühne, and R. Drechsler, "Analyzing functional coverage in bounded model checking," *IEEE Trans. on CAD*, vol. 27, pp. 1305–1314, July 2008.

[24] D. Große, R. Wille, U. Kühne, and R. Drechsler, "Contradictory antecedent debugging in bounded model checking," in *ACM Great Lakes Symposium on VLSI*, 2009.

[25] U. Kühne, D. Große, and R. Drechsler, "Property analysis and design understanding," in *Design, Automation and Test in Europe*, 2009.

[26] D. Brand, "Redundancy and don't cares in logic synthesis," *IEEE Trans. on Comp.*, vol. 32, no. 10, pp. 947–952, 1983.

[27] C. Pixley, "A theory and implementation of sequential hardware equivalence," *IEEE Trans. on CAD*, vol. 11, no. 12, pp. 1469–1478, 1992.

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

[29] 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.

[30] A. Sülflow, G. Fey, R. Bloem, and R. Drechsler, "Using unsatisfiable cores to debug multiple design errors," in *Great Lakes Symp. VLSI*, 2008, pp. 77–82.

[31] 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.

[32] T. Austin and V. Bertacco, "Deployment of better than worst-case design: Solutions and needs," in *Int'l Conf. on Comp. Design*, 2005, pp. 550–558.

[33] Q. Zhou and K. Mohanram, "Gate sizing to radiation harden combinational logic," *IEEE Trans. on CAD*, vol. 25, no. 1, pp. 155–166, 2006.

[34] G. Fey and R. Drechsler, "A basis for formal robustness checking," in *Int'l Symp. on Quality Electronic Design*, 2008, pp. 784–789.

[35] G. Fey, A. Sülflow, and R. Drechsler, "Computing bounds for fault tolerance using formal techniques," in *Design Automation Conf.*, 2009.

[36] N. Eén and N. Sörensson, "An extensible SAT solver," in *SAT 2003*, ser. LNCS, vol. 2919, 2004, pp. 502–518.

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

[38] A. Biere, *BooleForce 1.0*. Johannes Kepler University of Linz (JKU), Institute for Formal Models and Verification, 2006, also available at http://fmv.jku.at/booleforce.

[39] ——, "Picosat essentials," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 4, pp. 75–97, 2008.

[40] H. Sheini and K. Sakallah, "Pueblo: A hybrid pseudo-boolean SAT solver," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 2, pp. 165–189, 2006.

[41] M. Fahim 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.

[42] A. Biere, "Resolve and expand," in *Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT)*, ser. LNCS, vol. 3542, 2005, pp. 59–70.

[43] R. Brummayer and A. Biere, "Boolector 0.4," in *SMT-COMP: Satisfiability Modulo Theories Competition*, 2008, available at http://fmv.jku.at/boolector/.

[44] C. Barrett and C. Tinelli, "CVC3," in *Computer Aided Verification*, ser. LNCS, vol. 4590. Springer-Verlag, 2007, pp. 298–302.

[45] R. Wille, G. Fey, D. Große, S. Eggersglüß, and R. Drechsler, "SWORD: a SAT like prover using word level information," in *Int'l Conference on Very Large Scale Integration*, 2007, pp. 88–93.

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

[47] Organising Committee of the Second International Competition of CSP Solvers, "XML Representation of Constraint Networks (Version 2.0)," 2006, http://www.cril.univ-artois.fr/ lecoutre/research/tools/format2.pdf.

[48] S. Merchez, C. Lecoutre, and F. Boussemart, "AbsCon: A prototype to solve CSPs with abstraction," in *Int. Conference on Principles and Practice of Constraint Programming*, ser. LNCS, vol. 2239, 2001, pp. 730–744.

[49] S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras, "SMT techniques for fast predicate abstraction," in *Computer Aided Verification*, 2006, pp. 424–437.