Refinement Strategies for Verification Methods Based on Datapath Abstraction

Zaher S. Andraus, Mark H. Liffiton, and Karem A. Sakallah

Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122

{zandrawi,liffiton,karem}@eecs.umich.edu

Abstract—**In this paper we explore the application of Counterexample-Guided Abstraction Refinement (CEGAR) in the context of microprocessor correspondence checking. The approach utilizes automatic datapath abstraction augmented with automatic refinement based on 1) localization, 2) generalization, and 3) minimal unsatisfiable subset (MUS) extraction. We introduce several refinement strategies and empirically evaluate their effectiveness on a set of microprocessor benchmarks. The data suggest that localization, generalization, and MUS extraction from both the abstract *and* concrete models are essential for effective verification. Additionally, refinement tends to converge faster when multiple MUses are extracted in each iteration.**

I. Introduction

Counterexample-Guided Abstraction Refinement (CEGAR for short) has been shown to be an effective paradigm in a variety of hardware and software verification scenarios. Originally pioneered by Kurshan [16], it has since been adopted by several researchers as a powerful means for coping with verification complexity. The widespread use of such a paradigm hinges, however, on the automation of its abstraction *and* refinement phases. Without automation, CEGAR requires laborious user intervention to choose the right abstractions and refinements based on a detailed understanding of the intricate interactions among the components of the design being verified. Clarke et al. [9], Jain et al. [14], and Dill et al. [5] have successfully demonstrated the automation of abstraction and refinement in the context of model checking for safety properties of hardware and software systems. In particular, these approaches create a smaller abstract transition system from the underlying concrete transition system and iteratively refine it with the spurious counterexamples produced by the model checker. The approaches in [9] and [14] are additionally based on the extraction of unsatisfiability explanations derived from the infeasible counterexamples to provide stronger refinement of the abstract model and to significantly reduce the number of refinement iterations. All of these approaches are examples of *predicate abstraction* which, essentially, projects the concrete model onto a given set of relevant predicates to produce an abstraction suitable for model checking a given property. In contrast, Andraus et al. [2] describe a methodology for *datapath abstraction* that is particularly suited for equivalence checking. In their approach, datapath components in behavioral Verilog models are automatically abstracted to uninterpreted functions and predicates while refinement is performed manually using the ACL2 theorem prover [15].

The use of (near)-minimal explanations of unsatisfiability forms the basis of another class of abstraction methods. These include the work of Gupta et al. [12] and McMillan et al. [20] who employ "proof analysis" techniques to create an abstraction from an unsatisfiable concrete bounded model checking (BMC) instance of a given depth.

In this paper we explore the application of CEGAR in the context of microprocessor correspondence checking. The approach is based on automatic datapath abstraction as in [2] augmented with automatic refinement using minimal unsatisfiable subset (MUS) extraction. One of our main conclusions is the necessity of basing refinement on the extraction of MUSes from both the abstract *and* concrete models. Additionally, refinement tends to converge faster when multiple MUSes are extracted in each iteration. Finally, localization and generalization of the spurious counterexamples are shown to be crucial for fast convergence of the refinement iteration.

The rest of the paper is organized in 4 sections. Section II reviews the basic CEGAR algorithm and describes the various refinement strategies that can be deployed to enhance its performance. Section III briefly describes datapath abstraction and illustrates the abstraction and various refinement steps on a simple example. In Section IV we describe our implementation of these ideas in the Reveal system and discuss the effectiveness of the various refinement options in the verification of a sample benchmark. We conclude in Section V, with a recap of the paper's main contributions and suggestions for further work.

II. Refinement Strategies

A sketch of the basic counterexample-guided abstraction refinement methodology is shown in Algorithm 1. The detailed design is assumed to be characterized by a system of *concrete* constraints $\text{conc}(X) = \bigwedge_{1 \le i \le n} C_i(X)$ where X denotes a suitable vector of variables and

$C_i(X)$ is a Boolean *consistency* constraint that models a particular component in the design. For example, a 32-bit adder with inputs A, B, and output S would be represented by the constraint $C(A, B, S) = (S = A + B)$. The verification objective can also be expressed as a Boolean function of the design's variables. In general we will be concerned with equivalence between signal pairs, but any safety property can be handled similarly. Let $\text{prop}(X)$ denote the condition that we would like to check for on the design. The verification task can be expressed as showing that $\text{conc}(X) \to \text{prop}(X)$ is valid, i.e., that it is a tautology.

CEGAR starts, in line 1, with the construction of an abstraction $\text{abst}(X) = \bigwedge_{1 \le i \le n} A_i(X)$ such that $C_i(X) \to A_i(X)$. In other words, each constraint in abst(X) is a *relaxation* of a corresponding constraint in conc$(X)^1$. This type of abstraction, is sound, i.e., if prop(X) holds on abst(X), then it must also hold on conc(X) (line 4), but incomplete, i.e., prop(X) may be violated on abst(X) but still hold on conc(X)). Completeness is achieved by *refining* abst(X) to eliminate such cases. Specifically, if X^* denotes an assignment of values to variables such that

$$\text{abst}(X^*) \to \text{prop}(X^*) = 0 \text{ and } \text{conc}(X^*) = 0$$

then X^* represents a spurious counterexample that must be eliminated from the space of consistent assignments in the abstract model (line 9). Note that this process is guaranteed to converge assuming that the abstraction is finite and noting that it shrinks monotonically with each refinement iteration.

This basic version of CEGAR can be quite inefficient, requiring a large number of refinement iterations, since false counterexamples are eliminated one at a time. The improved version depicted in Algorithm 2 employs several techniques aimed at reducing the number of refinement iterations. The common goal of these techniques is to use the specific counterexample that falsifies the property on the abstract model as a seed to generate a large collection of "related" counterexamples that can then be simultaneously checked on the concrete model and, subsequently, eliminated at the same time from the abstract model. Algorithm 2 employs the following *violation enlargement* methods:

- **Localization** (line 5a) is basically a cone-of-influence (COI) reduction that removes irrelevant (don't-care) assignments from the counterexample by a syntactic traversal of the abstract formula.
- **Generalization** (line 5b) replaces the specific values of the variables in the counterexample with appropriate

---

Algorithm 1 (Basic CEGAR)
```
1.  abst(X) = relax(conc(X))
2.  while (true) {
3.    if abst(X) -> prop(X)
4.    then {"property holds"; exit}
5.    else {// abst(X*) -> prop(X*) == 0
6.      if conc(X*)
7.      then {"property fails"; exit}
8.      else // spurious counter example
9.        abst(X) = abst(X) && !X*;
10.   }// else(line 5)
11. }// while(line 2)
```

Algorithm 2 (CEGAR with enhanced refinement)
```
1.  abst(X) = relax(conc(X))
2.  while (true) {
3.    if abst(X) -> prop(X)
4.    then {"property holds"; exit}
5.    else {// abst(X*) -> prop(X*) = 0
5a.     viol(X) = localize(X*);
5b.     viol(X) = generalize(viol(X));
5c.     expl(X) = MUS(!abst(X) && viol(X));
5d.     viol(X) = viol(X) && expl(X);
6.      if conc(X) && viol(X)
7.      then {"property fails"; exit}
8.      else // spurious violation
8a.       while (expl(X) = newMUS(conc(X)&& viol(X)))
9.          abst(X) = abst(X) && !(viol(X) && expl(X));
10.   }// else(line 5)
11. }// while(line 2)
```

equality and dis-equality between pairs of variables. These relations can now be viewed as a conjunction of violation constraints viol(X) that satisfy

$$[\text{abst}(X) \to \text{prop}(X)] \wedge \text{viol}(X) = 0$$

- **Minimal Unsatisfiable Subset (MUS) extraction** (line 5c) selects a small subset of the constraints from the unsatisfiable formula $\neg\text{abst}(X) \wedge \text{viol}(X)^2$ that is still sufficient to explain its infeasibility. Denoting this subset as expl(X), the set of violation constraints is now reduced by eliminating from it those constraints that are not in expl(X) (line 5d).

At this point, viol(X) represents not just one but many specific counterexamples that violate prop(X) but are consistent with abst(X). The enlargement of X^* to viol(X) was based solely on information from the abstract model and the property being checked. The check in line 6 is now used to determine if any of these violations is consistent with the concrete model. Failing this check implies that *all* of these violations are spurious and we can refine the abstract model by removing them. However, an additional enhancement (lines 8a and 9) makes it possible to utilize the concrete constraints to enlarge viol(X) further. Specifically, a small number of MUSes from the unsatisfiable formula $\text{conc}(X) \wedge \text{viol}(X)$ are extracted and used to select subsets of viol(X) each of which is sufficient to keep the formula unsatisfiable.

---

[1] In general, both design variables and the concrete constraints that relate them are relaxed.

[2] Note that $[\text{abst}(X) \to \text{prop}(X)] \wedge \text{viol}(X) = 0$ implies that $\neg\text{abst}(X) \wedge \text{viol}(X) = 0$.

```
module example();
    wire [3:0] a, b;
    wire m = a[3]; // msb
    wire l = a[0]; // lsb
    wire c = m? a >> 1 : a;
    wire d = l? b >> 2 : c;
    wire e = m? a : a >> 1;
    wire f = l? {2'b00, b[3:2]} : e;
    wire p = !(a == 0) || (d == f);
endmodule;
```

Fig. 1. Verilog design example used to illustrate abstraction and refinement

III. DATAPATH ABSTRACTION REFINEMENT

The focus of our work is the application of the CEGAR framework, particularly the afore-mentioned enhanced refinement strategies, in the context of hardware correspondence checking. Specifically, we address the task of verifying that an optimized microprocessor implementation is compliant with its functional specification. The implementation and specification are assumed to be given using a hardware description language, such as Verilog, and together are regarded as the concrete model. The correctness criterion, i.e., the property that must hold to insure that the implementation is functionally equivalent to the specification, depends on the nature of the optimizations performed to obtain the implementation. In particular, pipelined implementations require alignment of the programmer-visible implementation and specification states which can be accomplished, for example, by flushing [8]. The specifics of the correctness criterion, while important, are orthogonal to the abstraction refinement flow and can be assumed, for our purposes, to be provided by the user along with the specification and implementation.

The concrete model is relaxed by treating datapath elements as uninterpreted functions (UFs) and uninterpreted predicates (UPs) that operate on unbounded terms [6]. This is justified by the fact that, generally speaking, design optimizations add significant complexity to an implementation's control logic while, mostly, preserving its datapath components. Datapath abstraction, thus, yields a compact representation that preserves the control interactions in the concrete model while maintaining functional consistency of the abstracted datapath elements. The resulting abstract model can now be viewed as a set of constraints—a formula—in CLU, a quantifier-free first-order logic with counter arithmetic and lambda expressions [6], and used in lieu of the detailed bit-level concrete model to check satisfaction of the desired correctness condition.

To illustrate the salient features of datapath abstraction refinement consider the example Verilog "design" in Fig. 1. The verification objective is to prove that signal p is always true, indicating that the design satisfies the condition $(a = 0) \rightarrow (d = f)$. The formula representing the concrete constraints of this design is written by inspection as

$$
\begin{aligned}
\text{conc}&(a,b,c,d,e,f,l,m,p) = \\
&(m = a[3]) \wedge \\
&(l = a[0]) \wedge \\
&(m \wedge (c = a >> 1) \vee \neg m \wedge (c = a)) \wedge \\
&(l \wedge (d = b >> 2) \vee \neg l \wedge (d = c)) \wedge \\
&(m \wedge (e = a) \vee \neg m \wedge (e = a >> 1)) \wedge \\
&(l \wedge (f = \{2\text{'b}00, b[3:2]\}) \vee \neg l \wedge (f = e)) \wedge \\
&(p = \neg(a = 0) \vee (d = f))
\end{aligned}
\tag{1}
$$

Using the semantics of bit vector operations, such as extraction, concatenation, and shifting, along with the standard Boolean connectives, this formula can be translated in a straightforward fashion to propositional conjunctive normal form (CNF) so that it can be checked for satisfiability by standard SAT solvers. In fact, for this simple example it is quite easy for a modern SAT solver to prove that $\text{conc} \wedge \neg p$ is unsatisfiable which is the same as saying that $\text{conc} \rightarrow p$ is valid.

Our objective, however, is to establish this result using CEGAR. A possible abstraction of this design is:

$$
\begin{aligned}
\text{abst}&(a,b,c,d,e,f,l,m,s,t,u,p,zero) = \\
&(m = \text{EX1}(a)) \wedge \\
&(l = \text{EX2}(a)) \wedge \\
&(s = \text{SR1}(a, \text{succ}(zero))) \wedge \\
&(t = \text{SR2}(b, \text{succ}(\text{succ}(zero)))) \wedge \\
&(u = \text{CT1}(zero, \text{EX3}(b))) \wedge \\
&(c = \text{ite}(m, s, a)) \wedge \\
&(d = \text{ite}(l, t, c)) \wedge \\
&(e = \text{ite}(m, a, s)) \wedge \\
&(f = \text{ite}(l, u, e)) \wedge \\
&(p = \neg \langle a = zero \rangle \vee (d = f))
\end{aligned}
\tag{2}
$$

where detailed bit vector operations have been replaced by UP and UF symbols. For example, EX1 is a UP that corresponds to extracting the most significant bit of a, and SR2 is a UF that corresponds to a right shift of b by two bits. Variables in this abstract formula that correspond to bit vectors in the concrete formula are now considered to be unbounded terms. They can be compared for equality to enforce functional consistency (given two terms t_1 and t_2 and a single-argument UF F, $(t_1 = t_2) \rightarrow (F(t_1) = F(t_2))$) but are otherwise uninterpreted having lost their concrete semantics. On the other hand, variables in the abstract formula that correspond to single bits in the concrete formula (such as m

21

TABLE I
EXECUTION TRACE OF ALGORITHM 2 ON EXAMPLE OF FIG. 1

	Iteration 1				Iteration 2			
	X^*	Localize	Generalize	Find MUS	X^*	Localize	Generalize	Find MUS
abst(X)	$a=0$	$a=0$	$a=0$	$a=0$	$a=0$	$a=0$	$a=0$	$a=0$
	$b=8$	$b=8$	$l=1$	$l=1$	$b=16$	$b=16$	$l=0$	$l=0$
	$c=16$		$t\neq u$	$t\neq u$	$c=8$	$c=8$	$m=1$	$t=u$
	$d=20$	$d=20$			$d=8$	$d=8$	$t=u$	$a\neq s$
	$e=0$				$e=0$	$e=0$	$a\neq s$	
	$f=12$	$f=12$			$f=0$	$f=0$		
	$l=1$	$l=1$			$l=0$	$l=0$		
	$m=1$				$m=1$	$m=1$		
	$s=16$				$s=8$	$s=8$		
	$t=20$	$t=20$			$t=3$	$t=3$		
	$u=12$	$u=12$			$u=3$	$u=3$		
conc(X)	$viol_1 = (a=0) \wedge (a[0]=1)$ $viol_2 = b\gg 2 \neq \{2\text{'b}00,b[3:2]\}$				$viol_3 = (a=0) \wedge (a \neq a\gg 1)$			

and l) retain their Boolean semantics and can be combined with the standard Boolean connectives. The remaining symbols in the formula represent the CLU built-in functions for counting (succ), decision (if-then-else or ite) and the smallest term (zero).

When Algorithm 2 is invoked on this example, it terminates with a proof of validity after two refinement iterations (see Table I). The counterexample produced in the first iteration is localized by eliminating irrelevant assignments, namely those that correspond to the "else" branches of the the ite operators involving variable l as well as other assignments that depend on them. Next, the remaining relevant assignments are generalized into the violation constraint $(a=0) \wedge (l=1) \wedge (t \neq u)$ which, in this case, cannot be enlarged further using MUS extraction from the abstract formula. Upon checking this violation constraint on the concrete formula it is found to be spurious, and leads to the creation of two simpler explanations: 1) the least significant bit of a cannot be 1 when a is 0, and 2) shifting b right by two bits is equivalent to concatenating zeros to the left of b's two most significant bits. The abstract formula is now refined to eliminate these two violations, i.e., at the start of the second iteration the correctness condition is checked against abst $\wedge \neg((a=0) \wedge (l=1)) \wedge \neg(t \neq u)$.

In the second iteration, localization is unable to eliminate any assignments from the counterexample. However, generalization retains only three of the assignments as well as the equality between t and u from the first iteration, and deduces that a and s are not equal. MUS extraction identifies that the assignment to m is immaterial to the current violation and can be safely removed. Checking this violation constraint on the concrete formula shows that it is still spurious, and identifies the

minimal explanation "shifting zeros yields zeros!" The concrete formula was thus able to remove the constraints $l=0$ and $t=u$ as irrelevant to the current violation. When this violation is eliminated, by refining with $\neg((a=0) \wedge (a \neq s))$, the algorithm terminates proving that p is always true.

IV. IMPLEMENTATION AND EXPERIMENTAL RESULTS

We implemented the above refinement strategies in the Reveal system which performs verification of hardware designs using datapath abstraction. Reveal consists of the following components:

- **Vapor** [2] for abstracting designs written in behavioral Verilog to the UCLID language.
- **UCLID** [6] for converting the abstracted design to a formula in the CLU logic.
- **Wave** for encoding the CLU formula in propositional logic, specifically in the conjunctive normal form suitable for SAT solvers [7].
- **zCore** and **zMinimal** [23] for extracting a single MUS from a CNF instance. To obtain an MUS from the abstract model, we convert the CLU expression in Algorithm 2 (line 5c) to a CNF instance using Ackerman encoding [1].
- **CAMUS** [19] for extracting one or more MUSes from a CNF instance.

We ran the experiments on an 64-bit 2.4-GHz AMD processor with 4GB of RAM running Linux.

To establish a baseline, we ran Algorithm 2 by disabling MUS extraction from both the abstract (line 5c) as well as the concrete (line 8a) formulas. In all cases, the procedure had to be aborted, even for simple designs, suggesting that MUS extraction is essential for refinement. We then ran another set of experiments in which localization and generalization were disabled forcing MUS extraction to be based on the specific counterexamples produced. Again, verification failed to finish in the allotted time in all tested cases, suggesting that localization and generalization are also essential for effective refinement.

We then performed a series of experiments using different combinations of MUS-based refinements with localization and generalization enabled. For ease of exposition, we will use refine(x, y) to denote refinement with:

- $x \in \{0,1\}$ MUSes extracted using the abstract formula
- $y \in \{0, 1, \text{some}, \text{all}\}$ MUSes extracted using the concrete formula

We used zCore and zMinimal to extract single MUSes, and CAMUS to extract multiple and all MUSes. The number of MUSes produced by CAMUS in the "multiple" mode can be controlled by a user-specified parameter and is typically between 3 and 7.

Table II shows the results of verifying two different properties on the PDLX benchmark [25] with various refinement combinations. This benchmark consists of 686 Verilog lines and 396 latches. In the first set of experi-

TABLE II
PDLX Refinement Results

	refine(0, y)			refine(1, y)		
	T	**I**	**M**	**T**	**I**	**M**
refine(x, 0)	T.O.	>9	N/A	T.O.	>7	>7
refine(x, 1)	T.O.	>37	>37	T.O.	>22	>22
refine(x, some)	T.O.	>7	>25	138	2	10
refine(x, all)	T.O.	>1	–	T.O.	>1	–

Buggy PDLX implementation; prop = ($\mathbf{RF_{Spec}} = \mathbf{RF_{Impl}}$)

	refine(0, y)			refine(1, y)		
	T	**I**	**M**	**T**	**I**	**M**
refine(x, 0)	T.O.	>13	N/A	T.O.	>10	N/A
refine(x, 1)	316	25	25	180	8	8
refine(x, some)	404	5	21	64	1	4
refine(x, all)	T.O.	>1	–	T.O.	>1	–

Bug-free PDLX implementation; prop = ($\mathbf{PC_{Spec}} = \mathbf{PC_{Impl}}$)

ments, the property being checked is the equivalence between the implementation and specification register files using a buggy implementation in which the ALU output is stuck at 0. In the second set of experiments, the correctness condition is the equivalence of the implementation and specification program counters; the implementation in these experiments was bug-free. Both of these criteria are based on the Burch and Dill [8] correspondence checking scheme of pipelined microprocessors. The columns in the table give the total verification time in seconds (T), the number of refinement iterations (I) and, where applicable, the number of MUSes extracted using the concrete model (M). A time out of 600 seconds was used in all experiments.

For the buggy design, the only refinement strategy that did not time out was refine(1, some). For the bug-free design, on the other hand, verification with refinements that employed one or several MUSes extracted from the concrete model finished within the allowed time. In both cases, however, the best performance was obtained with the refine(1, some) combination.

Further analysis of these results reveals that the use of MUSes involves a trade-off between the effort to extract them and their effectiveness in refining the abstraction. This can be seen by comparing refine(x, all), refine(x, 1) and refine(x, some). In the first case, the excessive time needed to extract all MUSes seems to negate their utility for refinement. Comparing the other two scenarios, we note that extracting several MUSes per refinement iteration seems to always yield fewer iterations, and shorter overall verification times, than does the extraction of just one MUS in each iteration.

V. Conclusions and Future Work

In this work we explored the use of MUS extraction for refinement of datapath abstractions in the CEGAR verification flow. We found that extraction of MUSes from both the abstract and concrete models is necessary for faster convergence. We also found that performance tends to improve when refinement is based on the extraction of a small number of MUSes, rather than a single MUS, from the concrete model in each iteration. Additionally, we introduced counterexample localization and generalization, and demonstrated their necessity for speeding up convergence of the refinement iteration.

The robustness and scalability of the verification framework described here can be enhanced further by incorporating several additional improvements. For example, faster MUS extraction from the abstract model may be possible if the extraction algorithm were to operate directly on the abstract CLU formula rather than on its lower-level CNF Boolean encoding. Additionally, extracting multiple MUSes from the abstract model, akin to the extraction of multiple MUSes from the concrete model, may yield a further reduction in the number of refienemnt iterations and, possibly, overall verification time. Finally, by analyzing the *structure* of the MUSes extracted from the concrete model, it may be possible to generalize them into *universal* rules (templates) that can be stored in a rule base that grows with the usage of the system and that can be consulted when verifying other designs.

References

[1] W. Ackerman, "Solvable Cases of the Decision Problem." North-Holland, Amsterdam, 1954.

[2] Z. S. Andraus and K. A. Sakallah, "Automatic Abstraction of Verilog Models", In Proceedings of 41st Design Automation Conference 2004, pp. 218-223.

[3] C. Barrett, D. Dill, and J. Levitt, "Validity checking for combinations of theories with equality". In FMCAD96, LNCS 1166, pp. 187-201.

[4] P. Chauhan, E. Clarke, J. Kukula, S. Sapra, H. Veith, and D. Wang, "Automated Abstraction Refinement for Model Checking Large State Spaces using SAT based Conflict Analysis", FMCAD02.

[5] S. Das and D. Dill, "Successive Approximation of Abstract Transition Relations" in 16th Annual IEEE Symposium on Logic in Computer Science (LICS) 2001.

[6] R. E. Bryant, S. K. Lahiri, S. A. Seshia, "Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions". In Proc. CAV, July 2002.

[7] R. E. Bryant, S. German, and M. N. Velev, "Exploiting

positive equality in a logic of equality with uninterpreted functions". ACM Transactions on Computational Logic, 2(1):93-134, January 2001.

[8] J. R. Burch and D. L. Dill, "Automatic Verification of Pipelined Microprocessor Control". CAV '94, D. L. Dill, ed., LNCS 818, Springer-Verlag, June 1994, pp. 68-80.

[9] E. Clarke, O. Grumberg. S. Jha, Y. Lu and H. Veith, "Counterexample-Guided Abstraction Refinement," In CAV 2000, pp. 154-169.

[10] N. Een and A. Biere, "Improved Subsumption Techniques for Variables Elimination in SAT," SAT 2005.

[11] S. Graf and H. Saidi, "Construction of abstract state graphs with PVS," In CAV, volume 1254, pp. 72-83, 1997.

[12] A. Gupta, M. Ganai, Z. Yang, and P. Ashar, "Iterative Abstraction Using SAT-based BMC with Proof Analysis." In Proc. of the International Conference on CAD, pp. 416-423, Nov. 2003.

[13] J. L. Hennessy and D. A. Patterson, "Computer Architecture: A Quantitative Approach", Morgan Kaufmann, 1990.

[14] H. Jain, D. Kroening and E. Clarke, "Predicate Abstraction and Verification of Verilog," Technical Report CMU-CS-04-139.

[15] M. Kaufmann and J. Moore, "An Industrial Strength Theorem Prover for a Logic Based on Common Lisp." IEEE Transactions on Software Engineering 23(4), April 1997, pp. 203-213.

[16] R. Kurshan, "Computer-Aided Verification of Coordinating Processes: The Automata-Theoritic Approach," Princeton University Press, 1999.

[17] D. Kroening, E. Clarke and K. Yorav, "Behavioral Consistency of C and Verilog Programs with Bounded Model Checking," in 40th DAC, 2003, pp. 368-342.

[18] S. K. Lahiri, S. A. Seshia, R. E. Bryant, "Modeling and Verification of Out-of-Order Microprocessors in UCLID", FMCAD 2002.

[19] M. H. Liffiton, M. D. Moffitt, M. E. Pollack, and K. A. Sakallah, "Identifying Conflicts in Overconstrained Temporal Problems," in *Proc. 19th International Joint Conference on Artificial Intelligence (IJCAI-05)*, pp. 205-211, Edinburgh, Scotland, 2005.

[20] K. L. McMillan and N. Amla, "Automatic Abstraction without Counterexamples." In International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'03), pp. 2-17, Warsaw, Poland, April, 2003, LNCS 2619.

[21] S. A. Seshia, S. K. Lahiri, R. E. Bryant, "A User's Guide to UCLID version 0.1".

[22] D. E. Thomas and P. R. Moorby, "The Verilog Hardware Description Language", Kluwer Academic Publishers, Nowell, Massachusetts, 1991.

[23] L. Zhang and S. Malik, "Extracting Small Unsatisfiable Cores from Unsatisfiable Boolean Formulas." In SAT, Springer-Verlag, 2003.

[24] www-2.cs.cmu.edu/~uclid

[25] vlsi.colorado.edu/~vis

24