Dynamic Abstraction Using SAT-based BMC *

Liang Zhang† Mukul R Prasad‡ Michael S Hsiao§ Thomas Sidle‡
†Cadence Design Systems, San Jose, CA
‡Advanced CAD Technology, Fujitsu Laboratories of America, Sunnyvale, CA
§Department of Electrical & Computer Engineering, Virginia Tech, Blacksburg, VA

ABSTRACT

We propose a new dynamic method of abstraction, which can be applied during successive steps of the model checking algorithm to further reduce the model produced by traditional static abstraction methods. This is facilitated by information gathered from an analysis of the proof of unsatisfiability of SAT-based bounded model checking problems formulated on the original design. The dynamic abstraction effectively allows the model checker to work with smaller abstract models. Experiments on several industrial benchmarks demonstrate that dynamic abstraction can significantly improve both the performance and the capacity of typical abstraction refinement flows.

Categories and Subject Descriptors B.5.2[Design Aids]:Verification

General Terms: Design, Verification

Keywords: Abstraction Refinement, Model Checking, SAT

1. INTRODUCTION

The application of model checking has traditionally been hampered by the commonly known *state explosion* problem. Abstraction refinement has recently emerged as a promising technology that has the potential to alleviate this problem.

The basic idea of *abstraction refinement* [7] is to verify the property at hand on a simplified version, or *abstraction*, of the given design. The abstraction is generated such that whenever a property passes on the abstract model, it is guaranteed to pass on the original design as well. However, when a property fails on the abstract model, the produced counter-examples must be checked to see if they are true counter-examples on the original design. If not, the model checking process is iterated with another abstract model which approximates the original model more closely. The new abstract model can be obtained either by refinement, which embellishes the current abstraction with more details from the original design [2, 5, 12] or by re-generating a more detailed abstract model from the original design [6, 11]. Usually the challenge in abstraction refinement is to construct as small an abstract model as possible so that the model checker can handle it easily. At the same time, the abstract model should retain sufficient details so that the model checker can prove the property.

Previous work on abstraction refinement based model checking uses *static abstraction* in that the abstract model produced by the abstraction step is never modified by the downstream model checker. In this work we make the key observation that when a property is checked on a circuit model, there may be state elements that are *partially abstractable*, *i.e.*, while a state element is necessary in the proof of the property, it may actually be required only in certain time-frames in the proof. For example, some latches in the design are solely present for initialization purposes.

We use this observation to develop a new method, we call it *dynamic abstraction*, whereby the initial abstract model of the design-under-verification is *further* abstracted during successive image computation steps of the model checking phase. Dynamic abstraction provides a more aggressive yet accurate abstraction methodology, effectively allowing the core model checking algorithm to work on smaller abstract models. Information regarding the dynamic abstractability of different latches is deduced from an analysis of the unsatisfiable core of SAT-based BMC problems formulated on the concrete model. The main contributions of this paper are the following:

- We introduce and develop the notion of the dynamic abstraction, which is orthogonal to static abstraction, which has traditionally been used in abstraction refinement frameworks.
- We propose two techniques for implementing dynamic abstraction in the context of a SAT-BMC based abstraction framework, which currently uses a BDD-based model checker.

Experiment of our initial implementations on large industrial designs demonstrates the effectiveness of proposed dynamic abstraction. However, it is important to note that the notion of dynamic abstraction goes beyond these specific implementations.

2. RELATED WORK

Abstraction refinement algorithms can be broadly classified into two categories: 1) counter-example driven and 2) counter-example independent. Counter-example driven methods [1, 4, 5, 10, 12] typically work by iteratively refining the current abstraction so as to block *a particular* false counter-example encountered in model checking the abstract model. The refinement algorithm could use a combination of structural heuristics and/or functional analysis based on SAT or BDDs. Recent papers [2, 9] enlarged the scope of the refinement by blocking multiple false counter-examples from the abstract model.

Counter-example independent abstraction refinement was introduced in [11] and also independently discovered in [6]. The basic idea is to perform a SAT-based BMC [3] for the property, up to some depth k, on the original design and then generate the abstract model based on an analysis of the *proof of unsatisfiability* [15] of the BMC problem. Since the abstraction preserves latches and gates that are included in the proof of unsatisfiability of the BMC problem, it guarantees that the abstract model does not have any counter-examples up to depth k. If needed, successive abstract models can be generated by solving BMC problems of increasing depths. The use of BMC to concretize abstract counter-examples was first proposed in [5]. Bjesse *et al.* [1] proposed

*This work was done when the first author was a PhD student at Virginia Tech.

DAC 2005, June 13–17, 2005, Anaheim, California, USA.

an enhancement of the concretization in that the concrete error trace dose not have to be in the same length as the abstract counter-example. Li *et al.* [8] proposed a new search strategy for the SAT solver so that the proof of unsatisfiability will generate smaller abstract models.

All of the above works share two common features: 1) the abstraction step is algorithmically distinct from the model checking phase, and 2) The abstraction is purely structural and has no temporal component, *i.e.* the same structural abstraction is used for each image computation step in BDD-based model checking.

Our dynamic abstraction can be distinguished from all previous abstraction algorithms based on these two aspects. Our method first analyzes the temporal behavior of various latches. Then, based on the analysis it dynamically abstracts away a set of latches *during* the course of the model checking. A key point is that dynamic abstraction can be applied in addition to any traditional abstraction method.

3. PRELIMINARIES

In this paper, we only consider model checking of invariants $\mathbf{AG}p$, where p is a boolean expression of the given circuit model. The circuit can be represented as $M = \langle T(X,Y,W), I(X)\rangle$, where W the set of inputs, X the set of present state variables, Y the set of next state variables, $T(X,Y,W)$ the transition relation, and $I(X)$ the set of initial states. M has a set of latches $L = \{l_1, l_2, \ldots, l_m\}$. x_i and y_i are the present state and next state variable corresponding to latch l_i. The transition relation T can be represented as:

$$T(X,Y,W) = \bigwedge_{i\in\{1\ldots m\}} T_i(X,y_i,W)$$

$T_i(X,y_i,W) = y_i \leftrightarrow \Delta_i(X,W)$ is the transition relation of latch l_i.

Given a subset of latches L_{abs} that we would like to abstract away from the design, the abstract model can be constructed by cutting open the feedback loop of latches L_{abs} at their present state variables X_{abs}.The abstract model can then be represented as $\widehat{M} = \langle \widehat{T}(\widehat{X},\widehat{Y},\widehat{W}), I(\widehat{X})\rangle$, where $\widehat{W} = W \cup X_{abs}$, $\widehat{X} = X - X_{abs}$, $\widehat{Y} = \{y_i : x_i \in \widehat{X}\}$.

The basic framework for abstraction refinement in our current implementation is similar to the one developed in [11] and [6]. A simplified version of the algorithm used in [11] is shown in Algorithm 3.1.

```
 1: k = InitValue
 2: if SAT-BMC(M, p, k) is SAT then
 3:     return "found error trace"
 4: else
 5:     Extract proof of unsatisfiability, P of SAT-BMC
 6:     M' = ABSTRACT(M, P)
 7: end if
 8: if MODEL_CHECK(M', p) returns PASS then
 9:     return "passing property"
10: else
11:     Increase bound k
12:     goto Step 2
13: end if
```

Algorithm 3.1: Abstraction Refinement Using SAT-BMC

A k-step BMC problem on the original model can be formulated by unrolling and replicating the transition relation T, k times. Let v be a variable in T, and $v^1, v^2, \ldots, v^k$ denote the k instantiations of v in the unrolled BMC problem. If the BMC problem is unsatisfiable (property holds on the original model), the SAT solver, such as [15], can produce a *proof of unsatisfiability* (POU) for it. The POU (denoted by $\mathcal{P}$ in the sequel) is essentially a small subset of CNF clauses from the original formula, which preserves (proves) the unsatisfiability of the original formula. The POU can be scanned to identify a set of latches L_{abs} such that for each $l \in L_{abs}$ the variables $l^1, l^2, \ldots, l^k$ do not appear in any clauses of the POU. These latches can then be abstracted away using aforementioned approach. The rationale is that since these latches provably do not contribute to the property check in the first k time-frames, they might be irrelevant in deciding this property for unbounded behaviors as well.

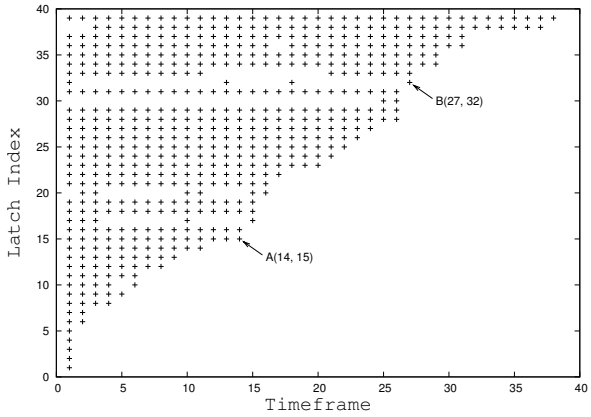

Figure 1: Latch-based Unsatisfiability Analysis

The model checking algorithm employed in Algorithm 3.1 (Step 8) may use a variety of methods, such as BDD-based symbolic model checker , SAT-based induction or BMC. Algorithm 3.2 shows pseudo code for a symbolic invariant checking algorithm using BDDs. Here B denotes the states that violate p and S_C denotes currently reached states. The core operation is the image operation *Img*, which computes the states reachable in one step, via the transition relation T, from the current states S_C, where $Img(Y) \equiv \exists X,W.\ S_C(X) \wedge T(X,Y,W)$.

```
InvariantCheck(M⟨T, I⟩, B)
 1: S_C = ∅; S_N = I;
 2: while S_C ≠ S_N do
 3:     S_C = S_N;
 4:     if B ∩ S_C ≠ ∅ then
 5:         return "found error trace";
 6:     end if
 7:     S_N = S_C ∪ Img(S_C);
 8: end while
 9: return "no bad state reachable";
```

Algorithm 3.2: Symbolic Invariant Checking

4. DYNAMIC ABSTRACTION

Given an original circuit M and the property $P = \mathbf{AG}p$, let us assume that a SAT-BMC problem on M of depth k has been solved and there is no counter-example. Further, suppose that the SAT solver generates a POU $\mathcal{P}$ for this problem. Figure 1 is a graphical representation of the POU from a 40-step SAT-BMC problem on a real circuit example. For each latch (plotted for 40 representative latches on the y-axis) the plot shows the time-frames at which the corresponding instantiation of the latch variable involves in the POU of the SAT-BMC problem. The latches have been sorted on the y-axis for better readability of the data. Given a latch variable $l \in L$, we can define the *redundancy index* (RI), $\rho(l)$ of l, with respect to the proof $\mathcal{P}$, as follows:

DEFINITION 4.1 (REDUNDANCY INDEX). *The redundancy index $\rho(l)$ of latch l with respect to the proof of unsatisfiability $\mathcal{P}$ is the smallest time-frame index such that for all time-frames j, $\rho(l) \le j \le k$, there does not exist a clause with variable l^j in $\mathcal{P}$.*

For example, in Figure 1 the points marked **A**(14,15) and **B**(27,32) show that the latch 15 has a RI of 15, while the latch 32 has a RI of 28. Simply put, the redundancy index is the earliest time-frame after which the given latch stops participating in the POU of the *corresponding* BMC problem. The situation depicted in Figure 1 is quite typical for a large variety of benchmarks we have experimented with. Large number of latches are not used in all time-frames of the POU. Moreover, some latches are *only* used in the first few time-frames.

At each step of image computation we can define a *candidate set* of latches which is essentially a set of latches whose redundancy index is no greater than the index of the current image computation step.

DEFINITION 4.2 (CANDIDATE SET). *The candidate set of latches for iteration j of image computation in Algorithm 3.2 is defined as $\mathcal{C}_j = \{l_i : l_i \in L, \rho(l_i) \leq j\}$*

For example, in Figure 1 the candidate set at time-frame 15 consists of the first 15 latches, *i.e.,* $\mathcal{C}_{15} = \{l_1, l_2, \ldots, l_{15}\}$. A modified version of Algorithm 3.2, incorporating dynamic abstraction is given below.

```
Invariant Check_DynamicAbstract (M⟨T, I⟩, B)
 1: S_C = ∅; S_N = I;
 2: while S_C ≠ S_N do
 3:    L_abs = CHOOSE_ABSTRACTION_LATCHES(L);
 4:    T = ABSTRACT_TR(L_abs, T);
 5:    B = ∃X_abs . B;
 6:    S_C = S_N;
 7:    S_C = ∃X_abs . S_C;
 8:    if B ∩ S_C ≠ ∅ then
 9:       return "found error trace";
10:    end if
11:    S_N = S_C ∪ Img(S_C);
12: end while
13: return "no bad state reachable";
```

Algorithm 4.1: Symbolic Invar. Checking with Dynamic Abstraction

In Algorithm 4.1 X_{abs} are the present state variables corresponding to the latches L_{abs} chosen for abstraction, ABSTRACT_TR abstracts the chosen latches from the T using the approach described in Section 3. The operation CHOOSE_ABSTRACTION_LATCHES may use different heuristics to choose a subset of the *candidate latches* $\mathcal{C}_j$ of the current iteration for abstraction. Since the dynamic abstraction is developed from an analysis of the POU of the k-step SAT-BMC problem, the following proposition about the Algorithm 4.1 is true.

THEOREM 4.1. *Algorithm 4.1 will not find any counter-example to the given property in the first k steps of image computation.*

The proof of of this theorem is along the lines of the main result in [11] Indeed, the static abstraction can be viewed as a special case of dynamic abstraction, since if we restrict our abstraction to latches with redundancy index '0', the Algorithm 4.1 becomes the conventional static abstraction-based invariant model checking.

4.1 Latch Selection Heuristic

A key component of Algorithm 4.1 is the latch selection heuristic CHOOSE_ABSTRACTION_LATCHES. This decides which latches, out of the current candidate set, should be abstracted at a given image computation step. This heuristic can have a significant bearing on the overall performance of the algorithm.

The most aggressive approach would be to perform dynamic abstraction for *all* latches in the current candidate set and at the earliest possible time as indicated by the redundancy index (RI) of each latch. However, this approach suffers from several drawbacks. The following issues drive the choice of this heuristic:

1. **How often to abstract latches:** Since the dynamic abstraction is implemented via quantification of next-state variables, from the transition relation, the overhead can be significant. Thus, a good heuristic should limit the abstraction to a few image computation steps.
2. **Extrapolating unbounded behavior from $\rho(l)$:** In abstraction based on POU of a k-step SAT-BMC we are trying to extrapolate unbounded behavior of a latch, with respect to the given property, based on its bounded behavior. Intuitively, a latch that was active only in the first few steps of the BMC (*i.e.* has a small RI), *e.g.*, latch l_2 in Figure 1, is more likely to be inactive beyond k time-steps than one which was active up to $k-1$ or $k-2$ steps (*i.e.* has a large RI), *e.g.*, latch l_{38} in Figure 1.
3. **Size and depth of the reachable state space:** Abstraction of latches comes at the cost of enlarging the set of permissible behaviors of the circuit. This can potentially enlarge the reachable state space, result in larger BDDs for the reached state representation and/or increase the depth of the reachability computation. This factor should be considered by the latch selection heuristic.

With the above criteria in mind, we have developed and tested several heuristics for CHOOSE_ABSTRACTION_LATCHES and found the following two to give a reasonable trade-off between overheads and abstraction power. Several other richer variants of these are possible and could be the subject of future research.

Heuristic 1: *Dynamically abstract just once at $\lceil \delta \cdot k \rceil$ time-steps, (where $0 < \delta < 1$), and abstract all latches in the candidate set at this point.*

The philosophy behind this heuristic to minimize the overheads of abstraction by doing it only once (issue 1 above) and being aggressive by choosing all candidates for abstraction. δ is kept fairly low to increase the likelihood of the latches being redundant for future image computations (in agreement with issue 2 above). Empirically, we used $\delta = 0.2$ in our experiments.

Heuristic 2: *Before the start of model checking analyze the proof $\mathcal{P}$ and gather a set of latches $\mathcal{S} = \{l : l \in L, \rho(l) \leq \delta \cdot k\}$. Every r steps of image computation, compute the set of latches $\mathcal{N}$* not *in the support set of current reached state set BDD. If $|\mathcal{S} \cap \mathcal{N}| \geq \tau$ abstract all latches in the set $\mathcal{S} \cap \mathcal{N}$. Repeat every r image computation steps.*

The intuition behind the using of the set $\mathcal{N}$ is that the removal of such latches is less likely to cause a blow-up in the current step of image computation. This ties in with issue 3 discussed above. Empirically, we used parameter settings of $\delta = 0.2$, $r = 2$, $\tau = 10$ for our experiments but the heuristic is not sensitive to these particular settings. Qualitatively, *Heuristic 1* is based on an aggressive one time application of dynamic abstraction whereas *Heuristic 2* is a more conservative and controlled application of dynamic abstraction. This distinction is born out by the experiments discussed in Section 5.

4.2 Handling Counter-examples

Our current implementation uses the counter-example independent refinement. Whenever a counter-example is found in the abstract model (line 9 of Algorithm 4.1) SAT-BMC is used to check if it is a true counter-example on the concrete model. If it is not a true counter-example, then we repeat the abstraction process with a deeper unrolling for SAT-BMC as shown in Algorithm 3.1. Otherwise, an error trace is returned to the user. Note that the effectiveness of proposed dynamic abstraction is unaffected by the underlying refinement scheme.

5. EXPERIMENTAL RESULTS

We have implemented the proposed dynamic abstraction algorithm as well as the static abstraction algorithm of [6, 11] within the VIS framework [13]. Our framework first abstracts a static model using an iterative abstraction loop. The POU extraction is based on the algorithm of [15] and it has been extended to report the redundancy index

Problem	Pass/ cex leng.	Concrete Model			Static Abstraction		Dynamic Heuristic I		Dynamic Heuristic II	
		# PIs	# FFs	# Gates	# FFs	Time(s)	# FFs (diff.)	Time(s)	# FFs (diff.)	Time(s)
P1	Pass	330	1158	5155	264	133	204 (-60)	**97**	221 (-43)	114
P2	Pass	401	1896	8910	257	936	218 (-39)	**189**	240 (-17)	550
P3	Pass	405	1951	8577	266	187	243 (-23)	141	252 (-14)	**172**
P4	Pass	671	2735	11381	271	**68**	211 (-60)	149	249 (-22)	**68**
P5	Pass	1015	2971	10044	286	474	216 (-70)	**260**	276 (-10)	441
P6	Pass	1020	3039	10060	277	360	224 (-53)	**92**	254 (-23)	141
P7	Pass	1981	5407	18193	245	322	222 (-23)	190	227 (-18)	**169**
P8	Pass	1950	5468	19161	224	1198	184 (-40)	**265**	201 (-23)	769
P9	Pass	1943	5644	19189	268	89	234 (-34)	87	247 (-21)	**79**
P10	Pass	3490	8998	27297	293	563	221 (-72)	**85**	269 (-24)	270
P11	60	308	746	3837	123	687	81 (-42)	**156**	87 (-36)	690
P12	36	289	654	4823	170	30704	160 (-10)	**1747**	170 (0)	30632
P13	29	289	654	4826	201	>24h (27)	168 (-33)	**28711**	196 (-5)	>24h (27)
P14	?	356	1644	7408	115	>24h (13)	93 (-22)	>24h (**26)**	109 (-6)	>24h (13)
P15	Pass	82	432	1740	146	7062	121(-25)	**3185**	135(-11)	5604

Table 1: Results: Static Abstraction and Proposed Dynamic Abstraction

(RI) for each latch. The downstream model checker has been modified to take RIs of latches as inputs. It then further abstracts the statically abstracted model on the fly using proposed latch selection heuristics. We use CUDD for the BDD-based computation, and ZCHAFF [14] as the SAT solver for BMC. We tested our tool for safety properties on different modules from four real-life industrial designs.

All experiments were run on 1.5 GHz Pentium 4 Linux machines with 1G RAM. The time-out limit is set to 24 hours.The results are reported in the Table 1. The second column shows if the property is a passing property or the length of the shortest counterexample. A question mark was shown for P14, since all methods timed out on it. Column 6 shows the number of latches in the statically abstracted model. Column 7 is the cumulative CPU time, which includes both abstraction and model checking time. Columns 8 and 10 report the number of latches in the final dynamically abstracted model for Heuristics 1 and 2 as well as additional latches abstracted with respect to the static abstraction method. For example, the static abstraction is able to abstract a model with only 224 latches for property P8, for which the concrete model has 5468 latches. Dynamic heuristic 1 is able to further abstract 40 more latches away, while dynamic heuristic 2 is able to abstract 23 more latches. The reported time in column 9 and 11 are also cumulative CPU times. For time-out cases, we also report the number of completed image computation steps. For example, for P14, dynamic heuristic 1 is able to compute 26 steps of images within 24 hours, while static abstraction and heuristic 2 can only finish 13 steps.

We can see that proposed heuristic 1 is extremely powerful in reducing the overall runtime, even though the number of additional abstracted latches may not be very significant. For example, with only 10 additional latches abstracted away for P12, it achieves over an order of magnitude speed-up compared to the pure static abstraction approach. However, as discussed in previous section, aggressive abstraction may potentially slow down the subsequent model checking. In our experiment, dynamic heuristic 1 does experience occasional slow-downs. As explained in the previous section, heuristic 2 is a more conservative and controlled application of the dynamic abstraction. It consistently outperforms the pure static abstraction for all cases significantly.

6. CONCLUSIONS

In this paper we have presented a method for performing dynamic abstraction within a framework for abstraction-refinement based model checking. The dynamic abstraction is applied during successive image computation steps of the model checking algorithm and can be applied in addition to traditional static abstraction methods. It is facilitated by information gathered from an analysis of the proof of unsatisfiability of SAT-based BMC problems formulated on the concrete model. We also proposed two strategies for realizing the dynamic abstraction. Our experiments on several large industrial designs demonstrate that proposed techniques can improve the performance of abstraction refinement based model checking by up to an order of magnitude compared to the state-of-the-art static abstraction refinement methods.

7. REFERENCES

[1] P. Bjesse and J. Kukula. Using Counter Example Guided Abstraction Refinement to Find Complex Bugs. In *Proc. of the Design Automation and Test in Europe Conf.*, pages 156–161, Feb. 2004.

[2] C. Wang et al. Improving Ariadne's Bundle by Following Multiple Threads in Abstraction Refinement. In *Proc. of the Intl. Conf. on CAD*, pages 408–415, Nov. 2003.

[3] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 19(1):7–34, July 2001. Kluwer.

[4] E. M. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT-based Abstraction Refinement Using ILP and Machine Learning Techniques. In *Intl. Conf. on Computer Aided Verification (CAV'02)*, LNCS 2404, pages 265–279, July 2002.

[5] D. Wang et al. Formal Property Verification by Abstraction Refinement with Formal, Simulation and Hybrid Engines. In *Proc. of the Design Automation Conf.*, pages 35–40, June 2001.

[6] A. Gupta, M. Ganai, Z. Yang, and P. Ashar. Iterative Abstraction Using SAT-based BMC with Proof Analysis. In *Proc. of the Intl. Conf. on CAD*, pages 416–423, Nov. 2003.

[7] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1995.

[8] B. Li and F. Somenzi. Efficient Computation of Small Abstraction Refinements. In *Proc. of the Intl. Conf. on CAD*, Nov. 2004.

[9] M. Glusman et al. Multiple-Counterexample Guided Iterative Abstraction Refinement: An Industrial Evaluation. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 176–191, April 2003.

[10] F. Y. C. Mang and P.-H. Ho. Abstraction Refinement by Controllability and Cooperativeness Analysis. In *Proc. of the Design Automation Conf.*, pages 224–229, June 2004.

[11] K. L. McMillan and N. Amla. Automatic abstraction without counterexamples. In *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 2619, pages 2–17, April 2003.

[12] P. Chauhan et al. Automated Abstraction Refinement for Model Checking Large State Spaces using SAT based Conflict Analysis. In *Formal Methods in Computer-Aided Design*, pages 33–51, Nov. 2002.

[13] R. Brayton et al. VIS: A system for Verification and Synthesis. In *Intl. Conf. on Computer Aided Verification*, pages 428–432, July 1996.

[14] http://ee.princeton.edu/˜chaff/zchaff.php, Dec. 2003.

[15] L. Zhang and S. Malik. Validating SAT Solvers using an Independent Resolution-based Checker: Practical Implementations and Other Applications. In *Proc. of the Design Automation and Test in Europe*, pages 880–885, March 2003.