Consistency Checking of All Different Constraints over Bit-Vectors within a SAT Solver

minor fixes marked blue

Armin Biere and Robert Brummayer
Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria
Email: {armin.biere,robert.brummayer}@jku.at

***Abstract*—This paper shows how all different constraints (ADCs) over bit-vectors can be handled within a SAT solver. It also contains encouraging experimental results in applying this technique to encode simple path constraints in bounded model checking. Finally, we present a new compact encoding of equalities and inequalities over bit-vectors in CNF.**

I. Introduction

Many applications require to reason about inequalities over bit-vectors. More specifically, one is often interested in constraining bit-vectors to be pairwise different. In SAT based bounded model checking [1] such all different constraints (ADCs) are used to model simple paths (loop-free paths), which are used to compute occurrence diameters (length of longest loop-free paths) [1], or reverse occurrence diameters for k-induction [2].

A straight-forward encoding of ADCs over bit-vectors to SAT is obviously quadratic in the number of bit-vectors. There are linear QBF encodings [3]–[5], but currently available QBF solvers have a hard time to take advantage of these more compact encodings. Non symbolic algorithms from constraint programming, such as [6], [7], are not applicable due to the large domain sizes of bit-vectors, e.g. a 32 bit-vector variable has a domain size of 2^{32} possible values.

In this paper we show how ADCs over bit-vectors can be embedded into a SAT solver. The technique is similar to the lazy approach in satisfiability modulo theories (SMT). See [8] for a recent survey on (lazy) SMT. In contrast to previous work [9], ADCs are checked inside the SAT solver. In our application ADCs are used to encode simple path constraints for k-induction [2].

We propose a new and effective way to extend a standard SAT solver to the theory of symbolic ADCs over bit-vectors. Our approach avoids costly restarts. Lemmas, generated for ADCs, are marked as learned clauses, which can be garbage collected. Furthermore, we present an efficient incremental consistency checking algorithm for symbolic all different constraints over bit-vectors. This technique allows us to solve many instances for which the classical approach [9] of incrementally adding bit-vector inequalities on demand fails. We always need less memory and also less time if we combine our technique with the refinement based approach [9].

II. External Consistency Checking of ADCs

Let $s_1, s_2, \ldots, s_m$ be bit-vectors of width n and $s_i(l)$ denote the l-th bit of bit-vector s_i. Then a simple but quadratic bit-level encoding of the ADC over the s_i is as follows:

$$\mathrm{adc}(s_1, \ldots, s_m) \;\equiv\; \bigwedge_{i<j} s_i \neq s_j \;\equiv\; \bigwedge_{i<j} \bigvee_{l=1}^{n} s_i(l) \oplus s_j(l)$$

where $\oplus$ denotes XOR. However, it turns out that in actual applications not all inequalities $s_i \neq s_j$ are required. Often, a small subset of inequalities is sufficient to conclude unsatisfiability. A standard technique to take advantage of this observation is to encode these inequalities lazily, and not eagerly [9], [10]. An inequality encoding is only added if the SAT solver returns a satisfying assignment that violates this particular inequality. In this case the SAT solver is restarted and the process continues until either a consistent assignment is found, or the SAT solver concludes the formula to be unsatisfiable. This technique is similar to an abstraction refinement loop in CEGAR [11].

However, in the worst case still quadratic many inequalities in m need to be encoded. Additionally, checking consistency outside of the SAT solver is expensive in this classical refinement based approach [9].

Consistency of a satisfying assignment returned by the SAT solver with respect to an ADC can be checked by either sorting the bit-vectors by their assigned values, or by hashing bit-vectors with their assigned values as key. In any case this check is linear in $m \cdot n$. Furthermore, the SAT solver has to be restarted. Finally, inconsistency can only be determined after the SAT solver generated a *full* assignment.

III. Internal Consistency Checking of ADCs

Our main contribution is a new consistency checking algorithm. It handles ADCs inside the SAT solver, which is an instance of Satisfiability Modulo Theories SMT.

To handle an ADC the SAT solver just needs its list of bit-vectors, which in turn are represented as lists of literals. These bit-vectors are added through the API in the same way as clauses. The size of the internal representation of these bit-vectors is linear in m, the number of bit-vectors in the ADC, and not quadratic in m as in an eager encoding. Internally, these bit-vectors are copied into all different objects (ADOs),

which contain a list of literals for the individual bits of the bit-vector, and a reference to the ADC they belong to.

One unassigned literal is watched in each ADO. This technique is similar to the two watching literal scheme of CHAFF [12], in which two unassigned literals per clause are watched. Whenever a watched literal becomes assigned, a new unassigned literal has to be found. If the search fails and all literals of an ADO are assigned, then the watched literal remains unchanged.

In the latter case all literals of the bit-vector are assigned. This concrete assignment to the bit-vector, represented by an ADO, is used as key to a hash table, associated with each ADC. In this table the ADOs of the ADC are stored. If another ADO with the same key is already in the hash table, an ADC inconsistency has been found. Otherwise, the fully instantiated ADO is entered into the hash table.

If a conflict occurs, a temporary clause of length $2 \cdot n$ is constructed. It contains the negation of all literals in the two bit-vectors, which have been assigned to equal values. This clause is a conflicting clause in the current assignment and is used as starting point for conflict analysis [13]. After a new learned clause is generated from this conflict, the temporary clause is discarded. Using a temporary clause avoids changing the procedure for analyzing the implication graph.

We should emphasize again, that neither the temporary clause is added to the CNF, nor a symbolic representation of the inequality of which this clause is an instance. Only clauses learned through conflict analysis starting from the temporary clause are added. We conjecture that similar clauses are learned as if inequalities are encoded symbolically.

During backtracking, ADOs are removed in reverse chronological order from the hash table. This can be implemented efficiently by saving the hash table position of the ADO together with the variable, whose assignment triggered the ADO to be added to the hash table. Whenever this variable becomes unassigned, the entry in the hash table is reset. Since entries to the ADO hash table are added and removed in a stack like fashion, also a hash table with open addressing can be used.

To simplify our implementation, we actually require that every variable occurs in at most one ADO. This restriction can always be enforced by adding copies of variables and enforcing equality through binary clauses. Nevertheless, the algorithm can be easily extended to variables resp. literals occurring in multiple ADOs and ADCs.

Updating watches for ADOs is cheaper than updating clauses of the corresponding eager encoding. Hashing the keys can be more costly, since it is linear in the bit-width n. The cost for entering and comparing keys can be ignored.

Watching two literals per ADO and changing the hash function slightly would also allow to derive forced assignments, which in the context of SMT is known as theory propagation [14]. However, this extension either requires to generate learned clauses for each such propagation, or major changes to the analysis function, which derives learned clauses. We leave the investigation of this extension as future work.

IV. ENCODING

Previous work, where ADCs are encoded either eagerly or lazily *outside* of the SAT solver, requires to encode a bit-vector inequality $s \neq t$, where s and t are bit-vectors of width n. We present a compact CNF encoding which up to our knowledge has not been described in the literature yet.[1] In order to encode $s \neq t$ to CNF, we introduce n fresh variables d_k:

$$\bigwedge_{k=0}^{n-1} \left((s_k \vee t_k \vee \bar{d}_k) \;\wedge\; (\bar{s}_k \vee \bar{t}_k \vee \bar{d}_k) \right)$$

The idea of this encoding is as follows. The variables d_k represent that s and t differ at position k. If $s_k = t_k$, then d_k is forced to *false*. However, if $s_k \neq t_k$, then d_k is unconstrained and can be set to *true*. Finally, we add the following *linking clause*, which enforces s and t to differ in at least one position:

$$\bigvee_{k=0}^{n-1} d_k$$

This idea can be extended to encode combinations of bit-vector inequalities and equalities, which have to be added in incremental refinement loops. Consider the following example:

$$i = j \;\Rightarrow\; v = w$$

This formula is an instance of Ackermann constraints [15] and can be used to enforce function congruence on demand, where i and j are unary function arguments, and v and w the results. First, we introduce a fresh variable e:

$$(i = j \;\Rightarrow\; e) \;\wedge\; (e \;\Rightarrow\; v = w)$$

This formula is equisatisfiable and can be rewritten into

$$(i \neq j \;\vee\; e) \;\wedge\; (\bar{e} \;\vee\; v = w)$$

Let n_1 be the number of bits of i and j, and let n_2 be the number of bits of v and w. As before, we introduce n_1 fresh variables d_k and encode $i \neq j$ as follows:

$$\bigwedge_{k=0}^{n_1-1} \left((i_k \vee j_k \vee \bar{d}_k) \;\wedge\; (\bar{i}_k \vee \bar{j}_k \vee \bar{d}_k) \right)$$

To encode $v = w$ we add the following clauses:

$$\bigwedge_{k=0}^{n_2-1} \left((\bar{v}_k \vee w_k \vee \bar{e}) \;\wedge\; (v_k \vee \bar{w}_k \vee \bar{e}) \right)$$

Finally, we relate the two parts through a *linking clause*:

$$e \vee \bigvee_{k=0}^{n_1-1} d_k$$

The idea of this encoding is as follows. If $i \neq j$, then they differ in at least one bit. Therefore, one d_k can be set to *true* to satisfy the linking clause. The variable e is now unconstrained and can be set to *false*. Therefore, v and w do not have to

[1] In our experiments this encoding is only used for eager and lazy encodings outside of the SAT solver, but not for our improved method for handling ADCs inside of the SAT solver.

TABLE I
OVERALL RESULTS

	partial	*complete*	*solved*	*unsolved*	*inconcl*	*sat*	*unsat*	*time* 10^3sec	*space* GB	*steps*
mixed	n	y	259	85	38	182	39	96	23.2	9736
refine	n	y	250	94	32	179	39	101	23.0	9698
sadc	n	y	244	100	36	171	37	103	17.2	9131
eager	n	y	242	102	27	177	38	102	31.9	9438
mixed	y	y	258	86	40	179	39	98	23.3	9792
sadc	y	y	243	101	33	172	38	104	17.1	9066
none	y	n	267	77	56	179	32	87	16.6	10877
base	y	n	283	61	96	187	0	70	28.9	15187

be equal. However, if $i = j$, all the d_k are forced to $false$. In order to satisfy the linking clause, e has to be $true$, which forces v and w to be equal.

V. EXPERIMENTS

We have built a simple SAT based bounded model checker MCAIGER.[2] It reads AIGER format [16] and uses PICOSAT [17] as back end. We have implemented the consistency checking algorithm described above in PICOSAT. PICOSAT provides an incremental API similar to MINISAT [9]. ADCs can be extended incrementally, by adding new bit-vectors.

MCAIGER follows [2], [9] to validate or falsify simple safety properties. It incrementally checks a *base case* and an *induction step* for increasing bounds. If the base case becomes satisfiable, the bad state is reachable. If the SAT instance of the induction step turns out to be unsatisfiable, then the bad state is unreachable. Otherwise, the new time frame is added, unless a limit on the number of steps is reached.

States are encoded as bit-vectors. Adding a new time frame incrementally extends the ADC of the states by adding the new state, unless no simple path constraints are used. Different strategies for enforcing simple paths through ADCs are discussed below.

In our experiments we used all the 344 benchmarks[3] of the Hardware Model Checking Competition in 2007 (HWMCC'07), which can be considered to be a representative set of model checking benchmarks. The setup is almost the same as in HWMCC'07: 900 seconds time limit, 1.5 GB memory limit. However, we only checked base and inductive case up to a bound of 100 steps. Therefore a benchmark is considered to be *solved*, if either

1) after at most 100 steps the base case is *satisfiable* and thus the bad state reachable,
2) after at most 100 steps the inductive case is *unsatisfiable*, i.e. the bad state can not be reached, **or**
3) the bound of 100 steps is reached without conclusive answer on the reachability of the bad state (*inconclusive*).

Thus, a benchmark is marked *unsolved* if during checking the base or inductive case the time or space limit is reached. [4]

[2]Available at http://fmv.jku.at/mcaiger
[3]Available at http://fmv.jku.at/hwmcc07
[4]The memory limit was only reached in two runs, where only the base case was checked. In all other cases *unsolved* instances are due to time out.

Table I summarizes the results. In the first column model checking algorithms are listed. The second and third columns show whether the algorithm uses simple path constraints in the base case and whether the algorithm is complete. The next five columns contain number of *solved*, *unsolved*, *inconclusive*, *satisfiable* (bad state reachable) and *unsatisfiable* (bad state proven not to be reachable) instances. The sum of the run times in seconds and the sum of the maximum memory in GB follow in the next two columns. Time and space outs contribute 900 seconds. The last column denotes the number of *steps* the algorithm was able to reach over all benchmarks.

The rows are partitioned into three parts. The lower part contains **base** case only checking, which is plain BMC without checking the inductive case, and **none**, which both checks the base and the inductive case. Both methods do not use simple path constraints and are thus incomplete. They do not solve the same problem as the other methods, but give a limit on what can be gained resp. lost by adding simple path constraints.

The upper part lists our new method **sadc**, which uses symbolic all different constraints handled in the SAT solver. It also contains the classical quadratic **eager** encoding of simple path constraints. The **refine** method [9] adds individual state inequalities as lemmas on demand [10] incrementally. Finally, **mixed** uses symbolic all different constraints as long the number of all different conflicts is small. If during one SAT call the number of conflicts due to all different constraints reaches a certain limit (1000 conflicts in these experiments) or the overall number of such conflicts in all calls to the SAT solver is above another limit (10000), then the model checker switches to the **refine** method.

Finally, the API of our SAT solver has been extended to temporarily disable detection of conflicts due to symbolic simple path constraints. This is useful for checking the base case only. In the inductive case consistency of symbolic simple path constraints is always enabled for complete methods. For **mixed** and **sadc** approach we list two experiments where we *partially* disable simple path constraints this way.

Symbolically handling all different constraints as in **sadc** needs much less memory. In its plain form it handles less examples than the **refine** approach of [9], but more than classical **eager** encoding. However, dynamically switching from **sadc** to **refine** as in the **mixed** approach can solve the largest number of examples.

To understand this result it is instructive to compare **sadc** with **refine** in more detail. The scatter plot of Fig. 1 reveals that there a lot of examples where **sadc** is much faster than **refine**, but also vice versa. These scatter plots use a double logarithmic scale. A cross marks the run time for **refine** on the vertical axis and for **sadc** on the horizontal axis. Thus, the region above the main diagonal shows runs where **sadc** is faster, below where **refine** is faster. The time limit of 900 seconds corresponds to the two light and dotted vertical resp. horizontal lines. The other diagonal dashed lines correspond to a factor of 2, 10, and 100 difference in run time.

Therefore, a combined approached can be beneficial. Even our simple strategy in **mixed**, which starts with **sadc** and

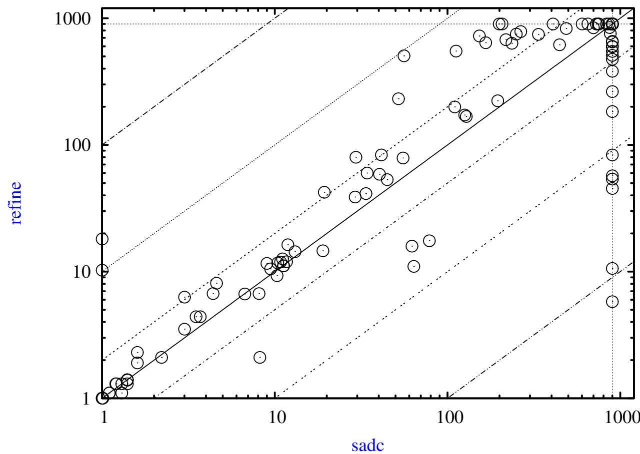

Fig. 1. **sadc** vs **refine**

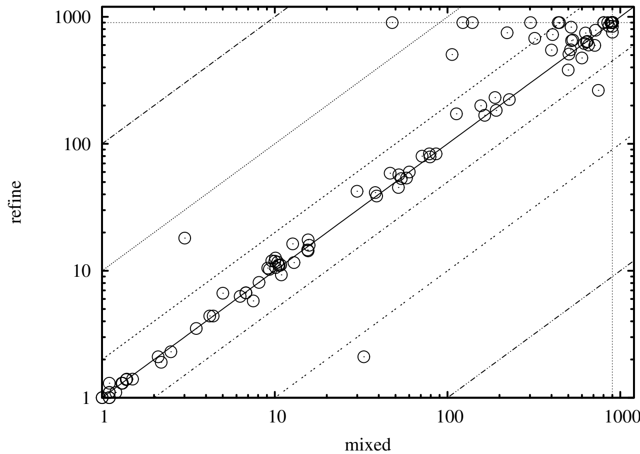

Fig. 2. **mixed** vs **refine**

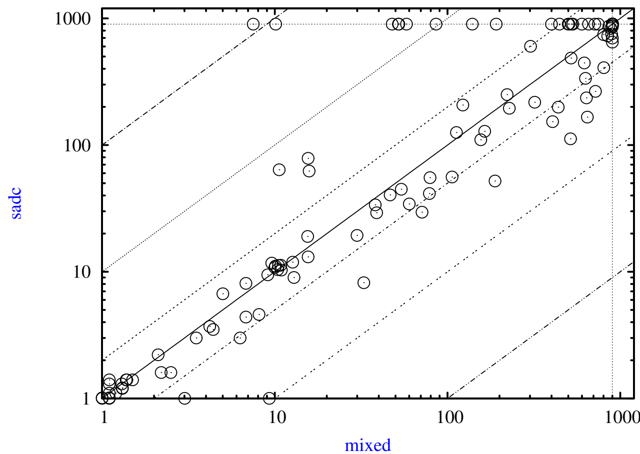

Fig. 3. **mixed** vs **sadc**

switches to **refine** as soon a conflict limit is reached seems to work as also the scatter plot of Fig. 2 and Fig. 3 show. The region above the main diagonal shows runs where **mixed** is faster than **refine** resp. **sadc**.

Tab. I also shows, that disabling simple path constraints temporarily in **sadc** or **refine** does not pay off. A more detailed analysis, not presented in this paper due to space constraints, reveals that the run-times are almost identical, except for a hand-full of benchmarks, where *partially* disabling simple path constraints hurts performance.

Finally, the reason that for some of the instances in the experiments, our new method performs worse than externally checking consistency, is most likely due the fact, that our current implementation does not propagate through ADCs.

VI. Conclusion

We have shown how all different constraints (ADCs) for bit-vectors can be handled inside a SAT solver symbolically. The technique does not require many changes to the SAT solver on the implementation side. Encouraging experimental results have been obtained.

So far we only check consistency of ADCs. In future work we want to use symbolic representations of ADCs for boolean constraint propagation (BCP) as well, which in the context of SMT is known as theory propagation [14]. We also think that it is worthwhile to apply similar techniques to other applications of equality logic over bit-vectors, such as encoding Ackerman constraints [15] for uninterpreted functions, or representing instances of McCarthy axioms for arrays [18].

References

[1] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, “Symbolic model checking without BDDs,” in *Proc. TACAS*, 1999.

[2] M. Sheeran, S. Singh, and G. Stålmarck, “Checking safety properties using induction and a SAT-solver,” in *Proc. FMCAD*, 2000.

[3] N. Dershowitz, Z. Hanna, and J. Katz, “Bounded model checking with QBF,” in *Proc. SAT*, 2005.

[4] T. Jussila and A. Biere, “Compressing BMC encodings with QBF,” in *Proc. BMC*, 2006.

[5] H. Mangassarian, A. Veneris, S. Safarpour, M. Benedetti, and D. Smith, “A performance-driven QBF-based iterative logic array representation with applications to verification, debug and test,” in *Proc. ICCAD*, 2007.

[6] J. Régin, “A filtering algorithm for constraints of difference in CSPs,” in *Proc. AAAI*, 1994.

[7] J. Marques-Silva and I. Lynce, “Towards robust CNF encodings of cardinality constraints,” in *Proc. CP*, 2007.

[8] R. Sebastiani, “Lazy satisfiability modulo theories,” *JSAT*, vol. 3, 2007.

[9] N. Eén and N. Sörensson, “Temporal induction by incremental SAT solving,” in *Proc. BMC*, 2003.

[10] L. de Moura and H. Rueß, “Lemmas on demand for satisfiability solvers,” in *Proc. SAT*, 2002.

[11] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, “Counterexample-guided abstraction refinement for symbolic model checking,” *J. ACM*, vol. 50, no. 5, 2003.

[12] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, “Chaff: Engineering an efficient SAT solver,” in *Proc. DAC*, 2001.

[13] J. Marques-Silva and K. Sakallah, “GRASP: A Search Algorithm for Propositional Satisfiability,” *IEEE T. on Computers*, vol. 48, no. 5, 1999.

[14] R. Nieuwnhuis and A. Oliveras, “DPLL(T) with exhaustive theory propagation and its applications to difference logic,” in *Proc. CAV*, 2005.

[15] W. Ackermann, “Solvable cases of the decision problem,” 1954.

[16] A. Biere, “AIGER And-Inverter Graph (AIG) format,” fmv.jku.at/aiger.

[17] ——, “PicoSAT essentials,” *JSAT*, 2008, submitted.

[18] J. McCarthy, “Towards a mathematical science of computation,” in *Proc. IFIP Congress*, 1962.