Lemmas on Demand for Satisfiability Solvers

Leonardo de Moura, and Harald Rueß

SRI International
Computer Science Laboratory
333 Ravenswood Avenue
Menlo Park, CA 94025, USA
{demoura, ruess}@csl.sri.com

Abstract. We investigate the combination of propositional SAT checkers with satisfiability procedures for domain-specific theories such as linear arithmetic, arrays, lists and the combination thereof. Our procedure realizes a lazy approach to satisfiability checking of Boolean constraint formulas by iteratively refining Boolean formulas based on lemmas generated on demand by the decision procedures.

1 Introduction

Boolean Satisfiability (SAT) have been successfully applied in different areas ranging from electronic design automation [7, 2] to artificial intelligence [6]. In bounded model checking [2] (BMC), for example, the system verification problem is encoded as a propositional formula such that *every* model of the formula corresponds to an invalid execution trace. In this way, BMC extends ideas for using SAT checkers to generate plans (as witnesses of eventually reaching goal) [6].

Finiteness is an inherent restriction in applications based on propositional representations such as BMC. Many applications, however, require data values from infinite domains such as integers, reals, arrays, or lists. Therefore, we investigate the combination of propositional SAT checkers with domain-specific satisfiability procedures as a conceptual basis for applications such as BMC over infinite domains. The basic idea is to iteratively refine Boolean formulas based on the analysis of a conjunction of constraints corresponding to a Boolean assignment. In this way, domain-specific knowledge is added lazily in terms of Boolean clauses.

For the special case of equality theories over terms with uninterpreted function symbols, Ackermann [1] already defined a reduction to Boolean logic by adding all possible applications of the congruence axiom. Variations of Ackermann's trick have been used, for example, by Pnueli et al [10] for validating compilation runs and by Bryant, German, and Velev [3] for equivalence checking of pipelined microprocessors. Compared to reductions based on Ackermann's translation, our procedure works for logics with a rich set of data types. Moreover, instead of constructing an equisatisfiable Boolean formula *a priori*, we compute a sequence of refinements by adding propositional lemmas as obtained from an analysis of spurious Boolean assignments. However, predetermined classes of lemmas may be used to speed up our convergence of our refinement process for certain applications. The process of refining Boolean formulas using lemmas generated on demand is similar to the refinement of abstractions based on the analysis of spurious counterexamples [4].

The paper is structured as follows. In Section 2 we provide some background material on satisfiability procedures and BMC. Section 3 describes our integrated approach and Section 4 summarizes some initial experience with a prototypical implementation of our integrated approach. Finally, in Section 5 we draw conclusions.

2 Background

Propositional variables can be assigned truth values *true* or *false*. A literal is a variable or its negation. A clause c is a disjunction of literals. A *CNF* formula is a conjunction of clauses. For our purposes, a SAT solver ($\mathcal{B}$-*sat*) is a function that receives a *CNF* formula and returns a truth assignment to the variables such that the formula becomes satisfied, or *unsatisfiable* if such assignment does not exist. A SAT solver is said to be incremental if the state of the procedure after processing φ can be used to decide if $\varphi \cup \varphi'$ is satisfiable.

A satisfiability procedure (constraint solver) $\mathcal{C}$-*sat* is a function that checks whether or not a set of constraints in a theory $\mathcal{C}$ is satisfiable. For instance, a linear programming system is a satisfiability solver for linear arithmetic.

Given a constraint theory $\mathcal{C}$, the set of *boolean constraints* $\mathsf{Bool}(\mathcal{C})$ includes all constraints in $\mathcal{C}$ and it is closed under conjunction $\wedge$, disjunction $\vee$, and negation $\neg$. The notions of satisfiability, inconsistency, satisfying assignment, and satisfiability solver are lifted to the set of boolean constraints in the usual way.

Bounded model checking (BMC) is concerned with searching for counterexamples of length k to the model checking problem $M \models \varphi$, where M is the system (program) being verified, and φ is a property. The approach is similar to planning as satisfiability, however, we have system transitions instead of actions, and the *goal* is a trace (counterexample) that falsifies φ.

3 The Integrated Satisfiability Solver

A satisfiability solver for $\mathsf{Bool}(\mathcal{C})$ can be obtained from a Boolean SAT solver and a $\mathcal{C}$-*solver* by simply converting the problem into disjunctive normal form, but the result is prohibitively expensive. Instead, we propose an algorithm based on the refinement of Boolean formulas with lemmas about constraints $c \in \mathcal{C}$. We restrict our analysis to formulas in CNF, since most SAT solvers expect their input to be in this format.

Translation schemes between propositional formulas and Boolean constraint formulas are needed, since SAT solvers and $\mathcal{C}$-*solvers* operate over disjoint sets of formulas. Given a formula φ such a corresponce is easily obtained by abstracting

procedure **sat**(φ)
  $p := \alpha(\varphi)$
  loop
    $\nu := \mathcal{B}\text{-}sat(p)$
    if $\nu = unsatisfiable$ **then** *unsatisfiable*
    else if $\mathcal{C}\text{-}sat(\gamma(\nu))$ **then** *satisfiable*
      else $p := p \cup create\text{-}lemmas(\nu)$

Fig. 1. Satisfiability solver for $\mathsf{Bool}(\mathcal{C})$

constraints in φ with (fresh) propositional variables. In this way, Let α be a funtion that maps constraints in $\mathcal{C}$ to propositional variables. This mapping induces a mapping from boolean constraint formulas to propositional formulas. For example, the formula $\varphi \equiv x_0 \geq 0 \wedge x_1 = x_0 + 1 \rightarrow x_1 \geq 1$ over linear arithmetic is mapped to $\alpha(\varphi) = p_1 \wedge p_2 \rightarrow p_3$, where $\alpha(x_0 \geq 0) \mapsto p_1$, $\alpha(x_1 = x_0 + 1) \mapsto p_2$, and $\alpha(x_1 \geq 1) \mapsto p_3$. Moreover, an assignment ν for propositional variables induces a set of constraints. Thus, let γ be the function that performs such mapping. For instance, the assignment $\nu = \{p_1 \mapsto \mathit{false}, p_2 \mapsto \mathit{true}, p_3 \mapsto \mathit{false}\}$ induces the set $\gamma(\nu) = \{x_0 < 0, x_1 = x_0 + 1, x_1 < 1\}$.

The procedure $\mathbf{sat}(\varphi)$ in Figure 1 combines a Boolean SAT solver $\mathcal{B}$-*sat* and a domain-specific satisfiability solver $\mathcal{C}$-*sat*. $\mathcal{B}$-*sat* generates a candidate Boolean assignment for $\alpha(\varphi)$. If there is no such candidate, the algorithm terminates, since φ is clearly unsatisfiable. Otherwise the satisfiability solver $\mathcal{C}$-*sat* is used to check whether or not the Boolean assignment ν determines a valid assignment for φ. If the assignment is not valid, new propositional clauses (*lemmas*) are added to the boolean formula at hand. The function *create-lemmas* is used to create such clauses. Optionally, *create-lemmas* uses the invalid assignment ν to guide the lemma generation.

A simple implementation of *create-lemmas* creates clauses of increasing size in each iteration. For example, if $\alpha(x_0 \geq 1) \mapsto p_1$, $\alpha(x_0 \geq 0) \mapsto p_2$, $\alpha(x_1 = x_0) \mapsto p_3$, $\alpha(x_1 \geq 1) \mapsto p_4$, $\alpha(x_1 \geq 0) \mapsto p_5$, the first call to *create-lemmas* produces the clauses $\neg p_1 \vee p_2$, and $\neg p_4 \vee p_5$, the second one produces the clauses $\neg p_1 \vee \neg p_3 \vee p_4$, $\neg p_1 \vee \neg p_3 \vee p_5$, and so on. This unguided enumeration is sound and complete procedure, but it is usually infeasible in practice, since the number of clauses of size k is $O(n^k)$, where n is the number of constraints.

Alternatively, clauses are added in a guided way based on the analysis of the set of constraints corresponding to a Boolean assignment. For instance, if the Boolean assignment $\nu = \{p_1 \mapsto \mathit{true}, p_2 \mapsto \mathit{false}, p_3 \mapsto \mathit{false}\}$ has been tested to yield an inconsistent set of constraints, the procedure *create-lemmas* adds the clause $\neg p_1 \vee p_2 \vee p_3$. This clause clearly prevents the invalid assignment to be regenerated by $\mathcal{B}$-*sat*. Therefore, the procedure of iteratively refining a Boolean formula based on the newly detected inconsistencies is terminating and complete. However, a naive implementation is also inefficient in practice, since only small fragments of the assignment ν are inconsistent. For example, suppose that an invalid assignment is associated with the following set of constraints:

$$\{x_0 \geq 0, y_0 \geq 0, x_1 = x_0, y_1 = y_0 + 1, x_2 = x_1 + 1, y_2 = y_1, x_2 \geq 1, x_2 < 1\}$$

It is clear that $\{x_2 \geq 1, x_2 < 1\}$ or $\{x_0 \geq 0, x_1 = x_0, x_2 = x_1 + 1, x_2 < 1\}$ are sufficient to describe the conflict. Therefore, let us assume that there is a function *explain* that returns an overapproximation of the minimal set of constraints that implies the inconsistency detected by $\mathcal{C}$-*sat*. This function is similar to the conflict resolution procedures found in SAT solvers such as GRASP [8] or Chaff [9]. Unfortunately, current implementations of domain-specific decision procedures lack such a conflict explanation facility *explain*. Therefore, we developed an algorithm that calls $\mathcal{C}$-*solver* $O(k \times n)$ times, where k is given, for finding such an overapproximation. An execution of this procedure is shown in Figure 2. First, since $\mathcal{C}$-*solver* detects the first conflict when processing the constraint $y_6 \leq 0$, this constraint is in the minimal inconsistent subset. Second, an overapproximation of the minimal set is produced by connecting constraints with common variables. Finally, an iterative approach is used to refine this overapproximation. In our example, two steps are sufficient to produce the *minimal* set $\{y_5 > 0, y_6 = y_5, y_6 \leq 0\}$. In general, this procedure is not only linear in the number of constraints but it also returns the exact minimal set if its size is less than or equal to k.

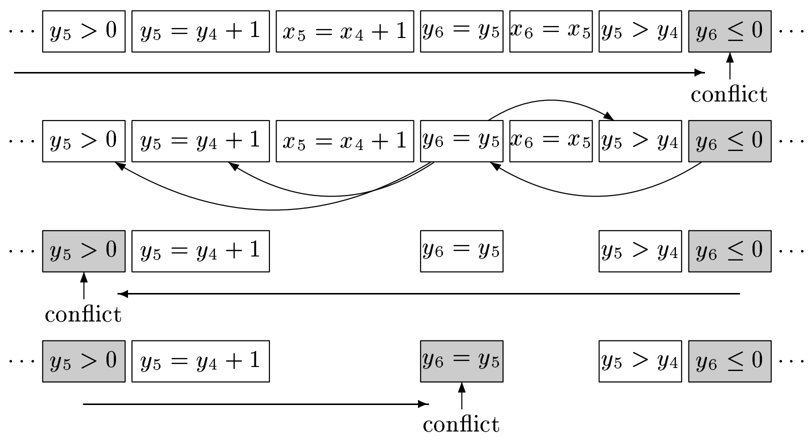

Fig. 2. Linear time *explain* function.

Now, we consider the BMC problem for systems with infinite state spaces. For systems of processes with an interleaving semantics only one action/transition occurs at every given step. Therefore for a given assignment only a small subset of constraints is needed to establish satisfiability. In other words, at step t it is only necessary to consider the constraints associated with the action/transition that occurs at step t. The constraints associated with other actions/transitions are *don't cares*. Overly eager assignments result in both useless search and overly specific plans/counterexamples. This can already be demonstrated using the abstract example $(q \wedge p_1) \oplus (\neg q \wedge p_2)$, where $\oplus$ is the *exclusive or*. Now, given an assignment $\nu = \{q \mapsto \mathit{true}, p_1 \mapsto \mathit{true}, p_2 \mapsto \mathit{true}\}$, suppose the following two situations:

1. $\alpha(p_1) \mapsto x \geq 0$, and $\alpha(p_2) \mapsto x = -1$, $\mathcal{C}$-$\mathit{sat}(\gamma(\nu))$ returns unsatisfiable, since $\{x \geq 0,\ x = -1\}$ is inconsistent. Therefore the assignment ν is discarded and the search continues. However, constraint p_2 is clearly irrelevant, that is, it is a *don't care*.
2. $\alpha(p_1) \mapsto x \geq 0$, and $\alpha(p_2) \mapsto x \leq 1$, $\mathcal{C}$-$\mathit{sat}(\gamma(\nu))$ returns satisfiable. However, the resultant model is overly specific, that is, the value of x is in the interval $[0, 1]$.

To ameliorate the situation we analyze the structure of the formula before CNF conversion. For each action/transition and step, we collect an auxiliary propositional variable that indicates whether the action/transition is executed at the given step or not. We use this information to assign *don't care* values to literals corresponding to inactive(unfired) actions(transitions) in each step. For example, the formula $(q \wedge p_1) \oplus (\neg q \wedge p_2)$ is mapped to the following CNF formula

$$\begin{aligned}
&(a_1 \vee a_2) \wedge (\neg a_1 \vee \neg a_2) \wedge \\
&(\neg q \vee \neg p_1 \vee a_1) \wedge (\neg a_1 \vee q) \wedge (\neg a_1 \vee p_1) \wedge \\
&(q \vee \neg p_2 \vee a_2) \wedge (\neg a_2 \vee \neg q) \wedge (\neg a_2 \vee p_2)
\end{aligned}$$

where, a_1 and a_2 are auxiliary propositional variables, that is, $a_1 \leftrightarrow q \wedge p_1$ and $a_2 \leftrightarrow \neg q \wedge p_2$. Now, given an assignment ν, the values of a_1 and a_2 are used to check whether the constraints p_1 and p_2 should be considered or not.

Finally, restarting the SAT solver in every loop is wasteful. However, if $\mathcal{B}$-sat is incremental, restarts can be avoided. Moreover, several SAT solvers use a conflict resolution procedure that records *conflict clauses* [8]. These clauses reduce the restart cost, since previously detected conflicts are avoided. However, not all conflicts can be avoided, since some of the conflict clauses are removed by the SAT solver to avoid exponential growth of the CNF formula.

4 Experiments

We implemented a prototypical satisfiability solver using Chaff [9] and ICS [5]. Its input is a Boolean formula in the standard DIMACS format extended with interpretations for propositional variables; for example, the line `i 7 |-> x >=y` assigns the constraint `x >= y` to the propositional variable with number `7`. The program either returns *unsatisfiable* in case the input Boolean constraint problem is unsatisfiable or an assignment for the variables. We describe some of our

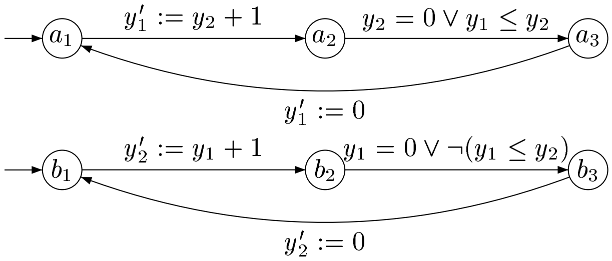

Fig. 3. Bakery Mutual Exclusion Protocol.

experiments using the Bakery mutual exclusion protocol in Figure 3.[1] with initial states $y_1 \geq 0 \wedge y_2 \geq 0$. The basic idea is that of a bakery, where customers take numbers, and whoever has the lowest number gets service next. Here, of course, "service" means entry to the critical section. In our example, there are only two processes (P_1 and P_2). The program location $a_3(b_3)$ represents the critical section of the process $P_1(P_2)$. The variable $y_1(y_2)$ contains the number that $P_1(P_2)$ uses to enter the critical section, it is zero if the process is not trying to enter the critical section. Only one process can execute a transition at each time. In this example, we want to verifiy whether both processes can be in the critical section at same time. The current variables are written as y_1 whereas the next-state variables are written as y_1'.

We use the notation x^i to represent the value of the variable x at time i. The variable $pc_1(pc_2)$ is the *program counter* of the process $P_1(P_2)$. Thus the formula that describes the initial state is:

$$pc_1^0 = a_1 \wedge y_1^0 \geq 0 \wedge pc_2^0 = b_1 \wedge y_2^0 \geq 0$$

We want to verify the property $\neg(pc_1 = a_3 \wedge pc_2 = b_3)$, thus, a counterexample of length k is a trace that reaches the *goal* $(pc_1^k = a_3 \wedge pc_2^k = b_3)$. The transitions are encoded as:

$$\begin{array}{l}
(pc_1^i = a_1 \wedge y_1^{i+1} = y_2^i + 1 \wedge pc_1^{i+1} = a_2 \wedge pc_2^{i+1} = pc_2^i \wedge y_2^{i+1} = y_2^i) \oplus \\
(pc_1^i = a_2 \wedge (y_2^i = 0 \vee y_1^i \leq y_2^i) \wedge y_1^{i+1} = y_1^i \wedge pc_1^{i+1} = a_3 \wedge pc_2^{i+1} = pc_2^i \wedge y_2^{i+1} = y_2^i) \oplus \\
(pc_1^i = a_3 \wedge y_1^{i+1} = 0 \wedge pc_1^{i+1} = a_1 \wedge pc_2^{i+1} = pc_2^i \wedge y_2^{i+1} = y_2^i) \oplus \\
(pc_2^i = b_1 \wedge y_2^{i+1} = y_1^i + 1 \wedge pc_2^{i+1} = b_2 \wedge pc_1^{i+1} = pc_1^i \wedge y_1^{i+1} = y_1^i) \oplus \\
(pc_2^i = b_2 \wedge (y_1^i = 0 \vee \neg(y_1^i \leq y_2^i)) \wedge y_2^{i+1} = y_2^i \wedge pc_2^{i+1} = b_3 \wedge pc_1^{i+1} = pc_1^i \wedge y_1^{i+1} = y_1^i) \oplus \\
(pc_2^i = b_3 \wedge y_2^{i+1} = 0 \wedge pc_2^{i+1} = b_1 \wedge pc_1^{i+1} = pc_1^i \wedge y_1^{i+1} = y_1^i)
\end{array}$$

This enconding includes the *frame axioms* to describe which variables a transition does *not* affect. The exclusive or ($\oplus$) operator guarantees that one and only one transition occurs at a time. The program counter (pc_1 and pc_2) can be encoded using propositional variables, since their domains are finite.

[1] See also www.csl.sri.com/~demoura/bmc-examples.

	don't cares, no explain	no don't cares, explain	don't cares, explain
depth	*new lemmas*	*new lemmas*	*new lemmas*
5	66	577	16
6	132	855	18
7	340	1405	58
8	710	1942	73
9	1297	2566	105
10	2296	-	185
15	-	-	646
20	-	-	1343

Table 1. Experimental Results.

Table 1 includes some statistics for three different configurations depending on whether *don't care* processing or the linear *explain* are disabled or not. In each column we list the number of additional lemmas (*new clauses*) necessary for detecting inconsistency of the problem at hand. This table indicates that the effort of assigning *don't care* values and the linear explain function are essential for efficiency.

Recall that the experiments so far represent worst-case scenarios in that the given formulas are unsatisfiable. For BMC problems with counterexamples, however, our procedure usually converges much faster. Consider, for example the mutual exclusion problem of the Bakery protocol with the assignment $y_2' := y_2+1$ instead of $y_2' := y_1 + 1$. The corresponding counterexample for $k = 7$ is produced in a fraction of a second after adding 53 lemmas.

$$\begin{array}{llll}
(a_1, k_1, b_1, k_2) & \rightarrow (a_1, k_1, b_2, 1 + k_2) & \rightarrow (a_2, 2 + k_2, b_2, 1 + k_2) & \rightarrow \\
(a_2, 2 + k_2, b_3, 1 + k_2) & \rightarrow (a_2, 2 + k_2, b_1, 0) & \rightarrow (a_3, 2 + k_2, b_1, 0) & \rightarrow \\
(a_3, 2 + k_2, b_2, 1) & \rightarrow (a_3, 2 + k_2, b_3, 1) & &
\end{array}$$

Notice that this counterexample represents a family of traces, since it is parametrized by (newly introduced constants) k_1 and k_2 with $k_1, k_2 \geq 0$.

5 Conclusion

We presented a generate-and-test approach to integrate boolean satisfiability checkers with domain-specific decision procedures. Boolean assignments generated by a SAT solver are tested by the constraint solver for consistency, and newly detected inconsistencies are used to iteratively refine Boolean formulas. We developed heuristics for accelerating this refinement process by computing good approximations of minimal inconsistent constraint sets at reasonable cost.

We barely scratched the surface of possible applications. For the rich logic of ICS—including constraints over uninterpreted function symbols, bitvectors, and arrays—our integrated solver is directly applicable to bounded model checking over large and infinite domains. It remains to be seen if our approach is also applicable to other application areas such as AI planners with resource constraints and domain-specific modeling.

References

1. W. Ackermann. Solvable cases of the decision problem. *Studies in Logic and the Foundation of Mathematics*, 1954.
2. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of bdds. In *Proceedings of DAC'99*, 1999.
3. R. E. Bryant, S. German, and M. N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In *Proceedings of CAV'99*, volume 1633 of *LNCS*, pages 470–482. Springer-Verlag, 1999.
4. Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proceedings of CAV'00*, volume 1855 of *LNCS*, pages 154–169, Chicago, IL, 2000. Springer-Verlag.
5. J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated Canonization and Solving. In *Proceedings of CAV'2001*, volume 2102 of *LNCS*, pages 246–249. Springer-Verlag, 2001.
6. Henry A. Kautz and Bart Selman. Planning as satisfiability. In *Proceedings of ECAI'92*, pages 359–363, 1992.
7. Tracy Larrabee. Test Pattern Generation Using Boolean Satisfiability. *IEEE Transactions on Computer-Aided Design*, 11(1):6–22, 1992.
8. Joao P. Marques-Silva and Karem A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *Proceedings of ICCAD'96*, pages 220–227, 1996.
9. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of DAC'01*, 2001.
10. A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small domains instantiations. In *Proceedings of CAV'99*, volume 1633 of *LNCS*, pages 455–469, Trento, Italy, 1999. Springer-Verlag.