

Lemmas on Demand for Satisfiability Solvers *

Leonardo de Moura and Harald Rueß

(`{demoura, ruess}@csl.sri.com`)

SRI International

Computer Science Laboratory

333 Ravenswood Avenue

Menlo Park, CA 94025, USA

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

1. Introduction

Many search and optimization problems can effectively be solved using propositional reasoning techniques. Finiteness, however, is an inherent restriction of propositional encodings, and computational systems and environment models are usually expressed more succinctly in logics enriched with domain-specific constraints. Planning problems in AI, for example, may involve solving numeric resource constraints, and program analyses often require reasoning about constraints in the combination of datatypes such as integers, arrays, lists, or bitvectors.

Given a decidable constraint theory, we address the problem of constructing effective solutions to the satisfiability problem for propositional combinations of constraints. Of course, satisfiability solvers for propositional constraint formulas can easily be obtained from the combination of a propositional SAT solver with decision procedures simply by converting the problem into disjunctive normal form, but the resulting algorithm is usually prohibitively expensive. Alternatively, propositional search capabilities can be added to theorem provers, but it seems to be more effective to augment propositional SAT solvers with theorem proving capabilities.

Here we look at the specific combination of SAT solvers with constraint solvers, and we propose a method that we call *lemmas on demand*, which invokes the constraint solver *lazily* in order to efficiently

* This research was supported by SRI International internal research and development, the DARPA NEST program through Contract F33615-01-C-1908 with AFRL, and the National Science Foundation under grants CCR-00-86096 and CCR-0082560.



prune out spurious counterexamples, namely, counterexamples that are generated by the SAT solver but discarded by the theorem prover by interpreting the propositional atoms. For example, the SAT solver might yield the satisfying assignment $p, \neg q$, where the propositional variable p represents the atom $x = y$, and q represents $f(x) = f(y)$. A decision procedure can easily detect the inconsistency in this assignment. More importantly, it can be used to generate a set of conflicting assignments that can be used to construct a lemma that further constrains the search. In the above example, the lemma $\neg p \vee q$ can be added as a new clause in the input to the SAT solver. This process of refining Boolean formulas is similar in spirit to the refinement of abstractions based on the analysis of spurious counterexamples or failed proof attempts [26, 25, 6, 16, 8, 14, 18].

From a set of inconsistent constraints in a spurious counterexample we obtain an *explanation* as an over-approximation of the minimal, inconsistent subset of these constraints. The smaller the explanation that is generated from a spurious counterexample, the greater the pruning in the subsequent search. In this way, the computation of explanations accelerates the convergence of our procedure.

The paper is structured as follows. Section 2 includes some background material, whereas Section 3 describes the *lemmas on demand* approach and various refinements thereof. Initial experience with this technique is reported in Section 4. Finally, in Section 6 we draw conclusions.

2. Background

We use the familiar concepts and notations of propositional logic and constraint logic. The truth values *true*, *false* are assigned to propositional variables. A literal is a propositional variable or its negation, a clause c is a disjunction of literals, and a *CNF* formula is a conjunction of clauses. There is a linear-time satisfiability-preserving transformation into CNF [22]. A propositional SAT solver (*B-sat*) is, for our purposes, a computable function that receives a *CNF* formula and returns either a satisfying truth assignment or *unsatisfiable* if such an assignment does not exist.

A (conjunctive) constraint solver, say *C-sat*, for a constraint theory \mathcal{C} , is a computable 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 constraint solver for linear arithmetic.

Given a constraint theory \mathcal{C} , the set of *Boolean constraints* $\text{Bool}(\mathcal{C})$ includes all constraints in \mathcal{C} and it is closed under conjunction \wedge ,

disjunction \vee , implication \rightarrow , 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.

Formulas in $\text{Bool}(\mathcal{C})$ can be translated into equisatisfiable Boolean formulas as long as the consistency of sets of constraints in \mathcal{C} is decidable. Translation schemes between propositional formulas and Boolean constraint formulas are needed. Given a formula φ such a correspondence is easily obtained by abstracting constraints in φ with (fresh) propositional variables. Let α be a function 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 $\nu = \{p_1 \mapsto \text{false}, p_2 \mapsto \text{true}, p_3 \mapsto \text{false}\}$ induces the set $\gamma(\nu) = \{x_0 < 0, x_1 = x_0 + 1, x_1 < 1\}$. Now, it is easy to see that a CNF formula φ in $\text{Bool}(\mathcal{C})$ is equisatisfiable with the Boolean formula (in CNF)

$$\alpha(\varphi) \wedge \left(\bigwedge_{\{l_1, \dots, l_n\} \in I(\varphi)} (\neg l_1 \vee \dots \vee \neg l_n) \right)$$

where $I(\varphi)$ is the set of subsets $\{l_1, \dots, l_n\}$ of literals l_i in $\alpha(\varphi)$ such that its “interpretation” $\{\gamma(l_1), \dots, \gamma(l_n)\}$ is inconsistent in \mathcal{C} . Thus, every $\text{Bool}(\mathcal{C})$ formula can be transformed into an equisatisfiable Boolean formula as long as there is a constraint solver for \mathcal{C} . On the other hand, the reduction seems to be infeasible, since an exponential number of \mathcal{C} -inconsistency checks is required in the worst case. It has been observed, however, that in many practical cases only small fragments of the set of \mathcal{C} -inconsistencies is needed. The main problem here is to identify small subsets of the set of all \mathcal{C} -inconsistencies which are sufficient to establish satisfiability of the Boolean constraint formula at hand.

3. Lemmas on Demand

We propose an algorithm based on the refinement of Boolean formulas with inconsistency lemmas that are generated on demand. We restrict ourselves to formulas in CNF, since most Boolean SAT solvers expect their input to be in this format.

The procedure $\text{sat}(\varphi)$ in Figure 3 combines a Boolean SAT solver $\mathcal{B}\text{-sat}$ and a domain-specific constraint solver $\mathcal{C}\text{-sat}$. $\mathcal{B}\text{-sat}$ generates a

```

procedure sat( $\varphi$ )
   $p := \alpha(\varphi)$ ;
  loop
     $\nu := \mathcal{B}\text{-sat}(p)$ ;
    if  $\nu = \text{unsatisfiable}$  then return unsatisfiable
    else if  $\mathcal{C}\text{-sat}(\gamma(\nu))$  then return satisfiable
    else  $p := p \cup \text{refine}(\nu)$ 

```

Figure 1. Lemmas on Demand for $\text{Bool}(\mathcal{C})$.

candidate Boolean assignment for $\alpha(\varphi)$. If there is no such candidate, the algorithm terminates, since φ is clearly unsatisfiable. Otherwise the satisfiability solver $\mathcal{C}\text{-sat}$ 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 (*inconsistency lemmas*) are added to the propositional formula at hand. The procedure *refine* is crucial in that it generates such new clauses. In order to guarantee soundness, all (interpretations of) clauses returned by *refine* are assumed to be implied by φ . In addition, the algorithm is *complete* if at least one clause returned by *refine* is not subsumed by clauses already in p . Alternatively, completeness can also be achieved by disabling infinite loops in which *refine* is only adding clauses subsumed by clauses already in p . We will return to a discussion about specific implementations of *refine* functions in Section 3.2.

3.1. CONSTRAINT THEORIES: EXAMPLES

One advantage of our approach is that it works uniformly for a large class of constraint theories, since the main requirement on these theories is the decidability of the conjunctive satisfiability problem. We review some of the more important constraint theories with polynomial satisfiability problem for the conjunction of a constraints. It follows that the satisfiability problem for the corresponding Boolean constraint theories are all NP-complete. In the following we assume as given a countably infinite set V of variables, and conjunctions of constraints are represented by finite sets.

3.1.1. Equality for Constants.

Satisfiability of conjunctive constraints C consisting of equalities $x = y$ and disequalities $x \neq y$ for variables x, y can be decided in linear time in the size of C . First, a graph is built, where the nodes are the variables and there is an edge between nodes x and y iff C contains the equality

$x = y$. Now, C is satisfiable iff for all $u \neq v$ in C it is the case that u and v are not connected in this graph.

3.1.2. Equality for Uninterpreted Functions.

Terms are either variables or applications $f(t_1, \dots, t_n)$, where f is a function symbol in some given signature of arity n . Satisfiability for a conjunction of equations and disequations over terms is decidable in $O(n \log(n))$ using congruence closure [11]. Satisfiability procedures for theories such as the one for *cons*, *car*, and *cdr* can be obtained using congruence closure algorithms [20] by adding all relevant instances of universally quantified axiom schemes such as $x = \text{car}(\text{cons}(x, y))$. Similarly, using Ackermann's trick [1] or a variation thereof, one can transform Boolean constraints over equalities for uninterpreted terms to an equisatisfiable Boolean problem with equations over variables as literals by adding all possible instances of the congruence axiom and renaming uninterpreted subterms with variables. In the worst case, the number of such axioms is proportional to the square of the length of the given formula.

3.1.3. Theories of Arithmetic.

Linear arithmetic constraints are built up from inequalities over linear arithmetic terms including rational constants and addition. When interpreted over the rationals, the conjunctive satisfiability problem for linear arithmetic constraints is polynomial, since it is equivalent to the linear programming problem, which is known to be polynomial; when interpreting linear arithmetic terms over the integers, the problem becomes NP-complete over the integers. The conjunctive satisfiability problem for nonlinear arithmetic constraints, which include also multiplication, is still decidable when interpreted over the rationals, but becomes undecidable over the integers. Pratt observed that most inequalities in program verification are of the form $x - y \leq c$, where c is constant. Think of a conjunction C of these constraints representing a directed graph whose nodes are labelled with variables and there is an edge from x to y of weight c for each constraint $x - y \leq c$. Now, C is satisfiable iff there exists a negative-weight cycle in this graph. Using then Bellman-Ford algorithm, satisfiability of C is decided in time quadratic to the number of variables in C . Shostak's [28] loop residue algorithm for linear constraints $a * x + b * y \leq c$ reduces to Pratt's algorithm when applied to difference constraints.

3.1.4. Theory of Fixed-Sized Bitvectors.

A core theory of equalities over fixed-sized bitvectors includes variables x_n , which are interpreted over bitvectors of width n , extraction $x_n[i : j]$ of bits i through j , and concatenation of two bitvector terms. From the results in [7] it follows that the conjunctive satisfiability problem for this theory is decidable in polynomial time when the width of variables and extraction positions are integer constants. These problems can easily be translated to equisatisfiable propositional SAT problems by bitwise splitting of the bitvector constraints. In practice, however, considerable performance gains have been reported by using domain-specific bitvector procedures instead of SAT solvers [15]. For example, a bitvector encoding of the shift register BMC benchmark is exponentially more succinct than the corresponding Boolean formula [10].

3.1.5. Combination of Satisfiability Procedures.

Many verification problems require to solve constraint problems in the union of constraint theories. There are two basic paradigms for combining decision procedures. The Nelson-Oppen [21] method combines decision procedures for disjoint theories by exchanging equality information on the shared variables. If the constituent decision procedures are polynomial, then the combined Nelson-Oppen procedure is polynomial, too. In Shostak's method [29, 24, 27] the combination of the theory of pure equality with canonizable and solvable theories is decided through an extension of congruence closure that yields a canonizer for the combined theory. Again, if the constituent canonizers and solvers are polynomial-time, then Shostak's algorithm also runs in polynomial time. All of the individual theories listed above can be combined using either the Nelson-Oppen or the Shostak approach. Consequently, satisfiability for propositional logic with constraints in the combination of any subset of these theories is NP-complete.

3.2. REFINEMENTS

Now we describe some possible implementations of the *refine* function in Figure 3. A simple implementation of *refine* 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 *refine* 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 a 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 $\nu = \{p_1 \mapsto \text{true}, p_2 \mapsto \text{false}, p_3 \mapsto \text{false}\}$ has been tested to yield an inconsistent set of constraints, the procedure *refine* adds the clause $\neg p_1 \vee p_2 \vee p_3$. This clause clearly prevents the invalid assignment to be regenerated by *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 over-approximation of the minimal set of constraints that implies the inconsistency detected by *C-sat*. This function is similar to the conflict resolution procedures found in Boolean SAT solvers such as GRASP [17] or Chaff [19]. Abstractly, conflict resolution procedures in Boolean SAT solver can be seen as a function that receives a *conflicting clause*¹ and returns a new clause that prevents this specific conflict in future iterations. These new clauses are called *conflict clause*, and the process of constructing them is sometimes referred to as *learning*. There is an obvious trade-off between the conciseness of this approximation and the cost for computing it. We are proposing an algorithm for finding such an over-approximation based on rerunning the constraint solver $O(m \times n)$ times, where m is some given upper bound on the number of iterations (see below) and n is the number of given constraints.

The run in Figure 3.2 illustrates this procedure. The constraints in Figure 3.2.(a) are asserted to *C-sat* from left-to-right. Since *C-sat* detects a conflict when asserting $y_6 \leq 0$, this constraint is in the minimal inconsistent set. Now, an over-approximation of the minimal inconsistent sets is produced by connecting constraints with common variables (Figure 3.2.(a)). This over-approximation is iteratively refined by collecting the constraints in an array as illustrated in Figure 3.2.(b). Configurations consist of triples (C, l, h) , where C is a set of constraints guaranteed to be in the minimal inconsistent set, and the integers l, h are the lower and upper bounds of constraint indices still under consideration. The initial configuration in our example is $(\{y_6 \leq 0\}, 0, 3)$. In each refinement step we maintain the invariant

¹ A conflicting clause is a clause in which all literals are assigned to *false*.

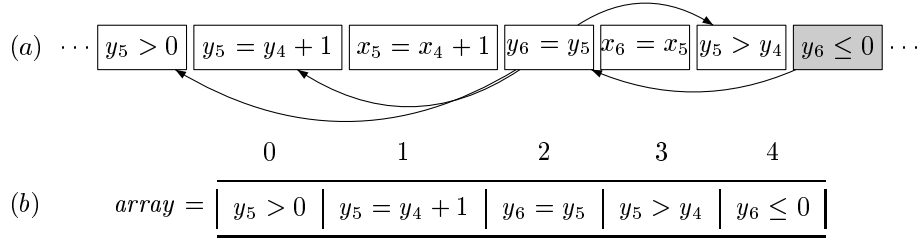


Figure 2. Trace for linear time *explain* function.

that $C \cup \{\text{array}[i] \mid l \leq i \leq h\}$ is inconsistent. Given a configuration (C, l, h) , individual constraints of index between l and h are added to C until an inconsistency is detected. In the first iteration of our running example we process constraints from right-to-left, and an inconsistency is only detected when processing $y_5 > 0$. The new configuration $(\{y_6 \leq 0, y_5 > 0\}, 1, 3)$ is obtained by adding this constraint to the set of constraints already known to be in a minimal inconsistent set, by leaving h unchanged, and by setting l to the increment of the index of the new constraint. The order in which constraints are asserted is inverted after each iteration. Thus, in the next step in our example, we successively add constraints between 1 and 3 from left-to-right to the set $\{y_6 \leq 0, y_5 > 0\}$. An inconsistency is first detected when asserting $y_6 = y_5$ to this set, and the new configuration is obtained as $(\{y_6 \leq 0, y_5 > 0, y_6 = y_5\}, 1, 1)$, since the lower bound l is now left unchanged and the upper bound is set to the decrement of the index of the constraint for which the inconsistency has been detected. The procedure terminates if C in the current configuration is inconsistent or after m refinements. In our example, two refinement steps yield the minimal inconsistent set $\{y_5 > 0, y_6 = y_5, y_6 \leq 0\}$. In general, the number of assertions is linear in the number of constraints, and the algorithm returns the exact minimal set if its cardinality is less than or equal to the upper bound m of iterations.

An additional refinement can be introduced in the procedure $\text{sat}(\varphi)$, since, in most cases, for a given assignment ν only a small subset of $\gamma(\nu)$ need to be considered. Overly eager assignments result in both useless search and overly specific counterexamples. For instance, assume the formula $(q \wedge p_1) \vee (\neg q \wedge p_2)$, and the assignment $\nu = \{q \mapsto \text{true}, p_1 \mapsto \text{true}, p_2 \mapsto \text{true}\}$, suppose the following two situations:

1. $\alpha(p_1) \mapsto x \geq 0$, and $\alpha(p_2) \mapsto x = -1$, $\mathcal{C}\text{-sat}(\gamma(\nu))$ 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*.

```

procedure collect( $f, \nu$ )
  if  $f \equiv c_1 \vee \dots \vee c_n$  then
    if  $f \in \gamma(\nu)$  then return collect(choose( $\{c_i \mid c_i \in \gamma(\nu)\}$ ),  $\nu$ )
    else return  $\bigcup_{i \in [1, n]}$  collect( $c_i, \nu$ )
  if  $f \equiv c_1 \Leftrightarrow c_2$  then
    return collect( $c_1, \nu$ )  $\cup$  collect( $c_2, \nu$ )
  if  $f \equiv \neg c$  then
    return collect( $c, \nu$ )
  if is-constraint( $f$ ) then
    if  $f \in \gamma(\nu)$  then return  $\{f\}$  else return  $\{\neg f\}$ 
  return  $\emptyset$ 

```

Figure 3. Collecting relevant constraints.

2. $\alpha(p_1) \mapsto x \geq 0$, and $\alpha(p_2) \mapsto x \leq 1$, $\mathcal{C}\text{-sat}(\gamma(\nu))$ returns satisfiable. However, the resulting set of models is overly specific in that the value of x is restricted to those in the interval $[0, 1]$.

To solve this problem we keep the structure of the formula before CNF translation. The structure of formula is used to decide whether a constraint is relevant in a given assignment or not. The procedure **collect**(f, ν) in Figure 3.2 collects all relevant constraints for a formula f and an assignment ν . For simplicity, this procedure only considers the propositional connectives: \vee , \Leftrightarrow , and \neg . The CNF translation adds a new propositional variable for each non-atomic sub-formula. It is important to notice that, $f \in \gamma(\nu)$ iff $\nu(\alpha(f)) = \text{true}$, that is, the formula f is assigned to *true* in the assignment ν . The function *is-constraint*(f) returns *true*, if f is a constraint. For instance, the formula $(q \wedge p_1) \vee (\neg q \wedge p_2)$ is represented as $\neg(\neg q \vee \neg p_1) \vee \neg(q \vee \neg p_2)$, and is translated to the following CNF formula:

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

where, a_1 and a_2 are auxiliary propositional variables, that is, $a_1 \equiv \neg(\neg q \vee \neg p_1)$ and $a_2 \equiv \neg(q \vee \neg p_2)$. Given an assignment $\nu = \{q \mapsto \text{true}, p_1 \mapsto \text{true}, p_2 \mapsto \text{true}, a_1 \mapsto \text{true}, a_2 \mapsto \text{false}\}$, it is clear that **collect**(f, ν) = $\{q, p_1\}$, that is, the value of p_2 is a *don't care*.

Figure 3.2 summarizes the guided *refinement* procedures discussed above. The procedure *refine-1* implements the *naive* approach without explanation capability and no specific consideration of *don't cares*. The

```

procedure refine-1( $\nu$ ) return { clausify( $\gamma(\nu)$ ) }
procedure refine-2( $\nu$ ) return { clausify(explain( $\gamma(\nu)$ )) }
procedure refine-3( $\nu$ ) return { clausify(collect( $\varphi, \nu$ )) }
procedure refine-4( $\nu$ ) return { clausify(explain(collect( $\varphi, \nu$ ))) }
procedure clausify( $C$ ) return {  $\alpha(l) \mid \exists l' \in C \wedge l = \neg l'$  }

```

Figure 4. Refinement functions.

```

procedure dpll()
  loop
    if decide() = done then return satisfiable
  loop
     $cc := bcp()$ ;
    if  $cc = nil$  then break
    if not conflict-resolution( $cc$ ) then
      return unsatisfiable

```

Figure 5. Davis-Putnam procedure.

procedure *clausify* converts a set of conflicting constraints to a clause. Procedure *refine-2* uses the explanation facility but no *don't cares*, whereas *refine-3* uses explanations and handles *don't cares* by collecting relevant constraints with *collect* in Figure 3.2. Finally, the procedure *refine-4* uses all optimizations described in this section.

3.3. ONLINE INTEGRATION

So far, we described an *offline* integration of *B-sat* and *C-sat*, in which the solvers are treated as black boxes, and both procedures are restarted in each refinement step. However, some *C-sat* tools support *backtracking*. In this case, an *online* integration is more appropriate, where choices for propositional variable assignments are synchronized with extending the logical context of the *C-sat* with the corresponding atoms. Detection of inconsistencies in the logical context of the *C-sat* triggers backtracking in the search for variable assignments. Furthermore, detected inconsistencies are propagated to the propositional search engine by adding the corresponding inconsistency clause (or, using an explanation function, a good over-approximation of the minimally inconsistent set of atoms in the logical context).

Figure 3.3 contains the main loop of the Davis-Putnam procedure found in most Boolean SAT solvers [17]. The algorithm starts with an

```

procedure C-dpll()
  loop
    status := decide();
    if status = done or should-propagate-to-C then
      cc := propagate-to-C();
      if cc ≠ nil then
        status := not-done;
        if not conflict-resolution(cc) then
          return unsatisfiable
      if status = done then return satisfiable
    loop
      cc := bcp();
      if cc = nil then break
      if not conflict-resolution(cc) then
        return unsatisfiable
procedure propagate-to-C()
  relevant-constrains := collect( $\varphi$ ,  $\nu$ );
  if C-assert(relevant-constrains) return nil
  return clausify(explain(relevant-constrains))

```

Figure 6. Online Integration.

empty boolean assignment, and traverses the space of truth assignments implicitly using a backtrack search algorithm. The search process iteratively performs the following steps: extends the current assignment by making a decision assignment to an unassigned variable (procedure *decide*); extends the current assignment by following logical consequences of the assignments made so far (procedure *bcp*), the deduction process may also identify and return a *conflicting clause* (variable *cc*), implying that the current assignment is not satisfiable; undoes (*backtracks*) the current assignment, if a conflict was detected, thus allowing another assignment to be tried (procedure *conflict-resolution*). The procedure *bcp* implements the boolean constraint propagation which corresponds to the application of the unit clause rule proposed by M. Davis and H. Putnam [9].

As described above, the *explain* function is similar to the conflict resolution procedure found in Boolean SAT solvers [17, 19]. Therefore, the conflict resolution procedure can be used to refine the result produced by the *explain* function in an online integration. For instance, suppose that the *explain* function returns the set $\{y > 10, y < 3\}$ as an explanation for a conflict detected by *C-sat*. Then, this set is

used to build the conflicting clause $\{\neg(y > 10), \neg(y < 3)\}$, which is sent to the conflict resolution procedure in \mathcal{B} -sat. A *conflict clause* is then produced by \mathcal{B} -sat. Figure 6 contains our *online* algorithm. The procedure *propagate-to-C* is responsible to send recently assigned constraints to \mathcal{C} -sat, it returns a conflicting clause if an inconsistency is detected. In other words, the procedure *propagate-to-C* implements the *bridge* between \mathcal{B} -sat and \mathcal{C} -sat. In our *online* algorithm, the procedure *collect* behaves slightly different, since it must handle unassigned variables, since ν can be a partial assignment. We also keep track of which constraints were already sent to \mathcal{C} -sat, so the procedure *collect* only collects the *unsent constraints*. The procedure *C-assert* is an incremental version of procedure *C-sat* in Figure 3, that is, it extends the logical context of \mathcal{C} -sat with the new constraints in the variable *relevant-constraints*. The procedure *propagate-to-C* is called when a satisfiable boolean assignment is found (*decide* returns *done*), or when the flag *should-propagate-to-C* is *active*. Different heuristics can be used to activate this flag, in our implementation it is activated every time a given number of new constraints are assigned to a boolean value. So, if the problem only contains propositional variables, our algorithm will behave like a standard Boolean SAT solver. Although it is not described in the Figure 6, the procedure *decide* must request \mathcal{C} -sat to create a new *backtracking point*, and *conflict-resolution* must request \mathcal{C} -sat to execute the *backtracking*. Our integrated algorithm is compatible with any kind of decision heuristic and standard optimizations such as *non-chronological backtracking* and *learning* [17].

4. Experiments

We implemented several refinements of the basic lazy theorem proving algorithm from Section 3, using Chaff [19] for the *offline* integration, and ICS [12] for deciding constraints. ICS is a ground decision procedure for the combination of linear arithmetic constraints, the theory of tuples, arrays, bitvectors, and equality over uninterpreted functions. Since state-of-the-art Boolean SAT solvers such as Chaff are missing the necessary API for realizing such an online integration, we used a home-grown SAT for realizing the online integration. We describe some of our experiments using the Bakery mutual exclusion protocol in Figure 4.² 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

² See also <http://www.csl.sri.com/~demoura/bmc-examples>.

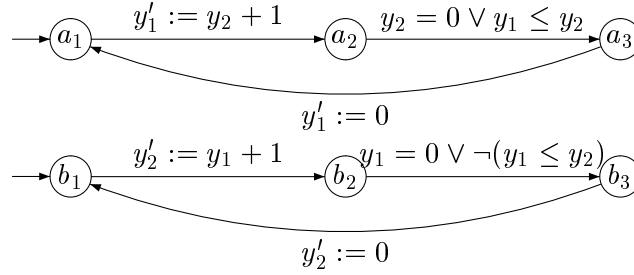


Figure 7. Bakery Mutual Exclusion Protocol.

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 are interested in the property that the processes are never in their critical sections at the same time. For validating this property we use bounded model checking (BMC) to search for counterexamples of length k to the model checking problem $M \models \varphi$, where M is the system (program) being verified, and φ is the mutual exclusion property. This technique has been introduced for finite systems in [4]. Here, we are working with an extension of the BMC methodology to infinite-state systems [10, 30].

We use the convention that current variables are always written as y_1, y_2 whereas the next-state variables are written as y'_1, y'_2 . In addition, x^i represents 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:

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

$$(pc_2^i = b_3 \wedge y_2^{i+1} = 0 \wedge pc_2^{i+1} = b_1 \wedge pc_1^{i+1} = pc_1^i \wedge y_1^{i+1} = y_1^i)$$

This encoding includes the *frame axioms* to describe which variables a transition does *not* affect. The program counter (pc_1 and pc_2) can be encoded using propositional variables, since their domains are finite.

Table I. Offline lazy theorem proving ('-' is time ≥ 1800 secs).

depth	<i>refine-2</i>		<i>refine-3</i>		<i>refine-4</i>	
	time	conflicts	time	conflicts	time	conflicts
5	45.23	577	0.71	66	0.31	16
6	83.32	855	2.36	132	0.32	18
7	286.81	1405	12.03	340	1.75	58
8	627.90	1942	56.65	710	2.90	73
9	1321.57	2566	230.88	1297	8.00	105
10	-	-	985.12	2296	15.28	185
15	-	-	-	-	511.12	646

Table I includes some statistics for three different *offline* configurations depending on which *refine* procedure described in Section 3 is used. For each configuration, we list the total time (in seconds) and the number of conflicts detected by the decision procedure. This table indicates that the effort of detecting the relevant constraints, 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.

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

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

The results of using this *online* integration for the Bakery example can be found in Table II for two different configurations.³ For each

³ The differences in the number of conflicts compared to Table I are due to the different heuristics of the SAT solvers used.

Table II. Online lazy theorem proving.

depth	no explain			explain		
	time	conflicts	calls to ICS	time	conflicts	calls to ICS
5	0.03	24	162	0.01	7	71
6	0.08	48	348	0.01	7	83
7	0.19	96	744	0.02	7	94
8	0.98	420	3426	0.05	29	461
9	2.78	936	7936	0.19	70	1205
10	8.60	2008	17567	0.26	85	1543
15	-	-	-	4.07	530	13468

configuration, we list the total time (in seconds), the number of conflicts detected by ICS, and the total number of calls to ICS. Altogether, using an explanation facility clearly pays off in that the number of refinement iterations (conflicts) is reduced considerable.

5. Related Work

For the special case of equality theories over terms with uninterpreted function symbols, Ackermann [1] already defined a reduction to Boolean logic by adding propositional encodings of all relevant instances of the congruence axiom. Variations of Ackermann's trick have been used, for example, by Shostak [28] for arithmetic reasoning in the presence of uninterpreted function symbols, and various reductions of the satisfiability problem of Boolean formulas over the theory of equality with uninterpreted function symbols to propositional SAT problems have recently been described by Goel, Sajid, Zhou, and Aziz [13], by Pnueli, Rodeh, Shtrichman, and Siegel [23], and by Bryant, German, and Velev [5]. In a similar vein, an eager reduction to propositional logic for constraints in Pratt's difference logic have been described by Strichman, Seshia, and Bryant [31]. Even for such a simple constraint theory, however, an exponential number of constraints may be generated in the preprocessing stage.

Compared with these *eager* reductions, our *lazy* integration procedure uniformly 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 propositional assignments. In this way, the semantics of constraints is introduced gradually and on

on demand. In this way, only inconsistency lemmas of relevance to the satisfiability of the formula are added.

In research that is most closely related to ours, Barrett, Dill, and Stump [3] describe an integration of Chaff with CVC by abstracting the Boolean constraint formula to a propositional approximation, then incrementally refining the approximation based on diagnosing conflicts using theorem proving, and finally adding the appropriate conflict clause to the propositional approximation. This integration corresponds directly to an online integration in the *lemmas on demand* paradigm. Their approach to generate good explanations is different from ours in that they extend CVC with a capability of abstract proofs for over-approximating minimal sets of inconsistencies. Also, optimizations based on *don't cares* are not considered explicitly in [3]. The experimental results in [3] coincide with ours in that they suggest that lazy theorem proving without explanations (there called the *naive* approach) and offline integration quickly become impractical. Using equivalence checking for pipelined microprocessors, speedups of several orders of magnitude over their earlier SVC system are obtained.

Armando, Castellini, and Giunchiglia [2] propose a SAT-based approach for the special case of solving disjunctions of Pratt's difference constraints. In their experiments, they observe excessively redundant computations, which can largely be eliminated using our explanation capabilities. A preprocessing step for computing inconsistency clauses with two literals is used [2] to simplify problems. We also found it to often to be advantageous to pregenerate 2- and even 3-inconsistencies to accelerate convergence. Optimizations based on *don't cares* are not considered in [2].

6. Conclusion

The main contribution of this paper is a *lazy* integration of propositional SAT solvers with constraint solvers for effectively deciding the satisfiability problem for propositional constraint formulas. The key idea is to use constraint solvers for suggesting, on demand, useful inconsistency lemmas. In this way, only inconsistency lemmas of relevance to the satisfiability of the formula are added. Various refinements such as online integration and acceleration of convergence using explanation functions are needed to make the *lemmas on demand* approach work effectively in practice.

References

1. W. Ackermann. Solvable cases of the decision problem. *Studies in Logic and the Foundation of Mathematics*, 1954.
2. A. Armando, C. Castellini, and E. Giunchiglia. SAT-based procedures for temporal reasoning. In *Proceedings of the Fifth European Conference on Planning (ECP-99)*, 1999.
3. C. W. Barrett, D. L. Dill, and A. Stump. Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT, 2002. To be presented at CAV 2002.
4. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proceedings of Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99)*, number 1579 in *LNCS*, 1999.
5. 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.
6. 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.
7. David Cyrluk, Harald Rueß, and Oliver Möller. An efficient decision procedure for the theory of fixed-sized bit-vectors. In *Computer-Aided Verification, CAV '97*, volume 1254 of *LNCS*, pages 60–71, Haifa, Israel, 1997. Springer-Verlag.
8. Satyaki Das and David L. Dill. Successive approximation of abstract transition relations. In *Symposium on Logic in Computer Science*, pages 51–60. IEEE, 2001.
9. Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.
10. Leonardo de Moura, Harald Rueß, and Maria Sorea. Lazy theorem proving for bounded model checking over infinite domains. In A. Voronkov, editor, *International Conference on Automated Deduction (CADE'02)*, Lecture Notes in Computer Science, Copenhagen, Denmark, July 2002. Springer-Verlag.
11. P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpressions problem. *Journal of the ACM*, 27(4):758–771, October 1980.
12. J.-C. Filiâtre, 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.
13. A. Goel, K. Sajid, H. Zhou, and A. Aziz. BDD based procedures for a theory of equality with uninterpreted functions. In *Proceedings of CAV'98*, volume 1427 of *LNCS*, pages 244–255. Springer-Verlag, 1998.
14. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. *ACM SIGPLAN Notices*, 31(1):58–70, 2002.
15. P. Johannsen and R. Drechsler. Formal verification on register transfer level—utilizing high-level information for hardware verification. In *IFIP International Conference on Very Large Scale Integration (VLSI'01)*, pages 127–132, Montpellier, France, 2001.
16. Yassine Lakhnech, Saddek Bensalem, Sergey Berezin, and Sam Owre. Incremental verification by abstraction. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 7th International Conference, TACAS 2001*, volume 2031 of *Lecture Notes in Computer Science*, pages 98–112, Genova, Italy, April 2001. Springer-Verlag.

17. Joao P. Marques-Silva and Karem A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *Proceedings of ICCAD'96*, pages 220–227, 1996.
18. M. Oliver Möller, Harald Rueß, and Maria Sorea. Predicate abstraction for dense real-time systems. *Electronic Notes in Theoretical Computer Science*, 65(2), 2002.
19. 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.
20. G. Nelson and D. Oppen. Fast decision algorithms based on congruence closure. Technical Report STAN-CS-77-646, Computer Science Department, Stanford University, 1977.
21. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.
22. David A. Plaisted and Steven Greenbaum. A structure preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986.
23. 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.
24. Harald Rueß and Natarajan Shankar. Deconstructing Shostak. In *16th Annual IEEE Symposium on Logic in Computer Science*, pages 19–28, Boston, MA, July 2001. IEEE Computer Society.
25. Vlad Rusu and Eli Singerman. On proving safety properties by integrating static analysis, theorem proving and abstraction. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 178–192, Amsterdam, The Netherlands, March 1999. Springer-Verlag.
26. H. Saïdi. Modular and incremental analysis of concurrent software systems. In *14th IEEE International Conference on Automated Software Engineering*, pages 92–101. IEEE Computer Society Press, 1999.
27. N. Shankar and Harald Rueß. Combining shostak theories. Invited paper for Floc'02/RTA'02, 2002.
28. Robert E. Shostak. Deciding linear inequalities by computing loop residues. *Journal of the ACM*, 28(4):769–779, October 1981.
29. Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.
30. Maria Sorea. Bounded model checking for timed automata. *Electronic Notes in Theoretical Computer Science*, 68(5), 2002.
31. O. Strichman, S.A. Seshia, and R.E. Bryant. Deciding Separation Formulas with SAT. In *International Conference on Computer-Aided Verification (CAV'02)*. Springer Verlag, July 2002.