A Scalable Method for Solving Satisfiability of Integer Linear Arithmetic Logic

Hossein M. Sheini and Karem A. Sakallah

Dept. of Electrical Engineering and Computer Science,
1301 Beal Avenue, Ann Arbor, MI 48109-2122, USA
Tel: +1 (734) 996-2528. Fax: +1 (734) 763-4617
{hsheini, karem}@umich.edu

Abstract. In this paper, we present a hybrid method for deciding problems involving integer and Boolean variables which is based on generic SAT solving techniques augmented with a) a polynomial-time ILP solver for the special class of Unit-Two-Variable-Per-Inequality (unit TVPI or UTVPI) constraints and b) an independent solver for general integer linear constraints. In our approach, we present a novel method for encoding linear constraints into the SAT solver through binary "indicator" variables. The hybrid SAT problem is subsequently solved using a SAT search procedure in close collaboration with the UTVPI solver. The UTVPI solver interacts closely with the Boolean SAT solver by passing implications and conflicting assignments. The non-UTVPI constraints are handled separately and participate in the learning scheme of the SAT solver through an innovative method based on the theory of cutting planes. Empirical evidence on software verification benchmarks is presented that demonstrates the advantages of our combined method.

1 Introduction

Many applications in hardware and software verification, such as RTL datapath verification [7], symbolic timing verification [2], and buffer over-run vulnerability detection [32], are naturally cast as decision problems that involve systems of constraints over the Booleans and unbounded integers. In the past several years, there has been considerable progress in solving these types of problems by leveraging the recent advances in Boolean SAT and combining them with methods that decide the feasibility of systems of linear integer constraints.

In this paper we are concerned with quantifier-free *Mixed Integer Boolean* formulas (MIB formulas for short) whose atoms are a) Boolean constants and variables, and b) linear integer constraints. Any such atom is a valid MIB formula, and if φ_1 and φ_2 are valid MIB formulas then so are $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, and $\varphi_1 \vee \varphi_2$. We note that a MIB formula reduces to a quantifier-free Presburger (QFP) formula when the set of Boolean atoms is empty. A MIB formula is said to be in *Conjunctive Normal Form* (MIB-CNF) if it is the conjunction of a set of *clauses* each of which is the disjunction of a set of *liter-*

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 241–256, 2005.

als where a literal is either an atom or the negation of an atom. An arbitrary MIB formula can be expressed in MIB-CNF in linear time and space by introducing a set of Boolean auxiliary variables, if necessary [33]. When the distinctions are not important, we will use the terms MIB formula and MIB-CNF formula in the sequel to generically refer to a QFP formula.

Current methods for deciding the satisfiability/unsatisfiability of MIB-CNF formulas can be divided into four categories:

1. **SAT-Based Abstraction/refinement methods** [4, 20]**.** In these approaches, the MIB-CNF formula φ is *abstracted* to a pure Boolean CNF formula $\hat{\varphi}$ by replacing each linear integer constraint with an unrestricted Boolean *indicator* variable. A SAT solver is then used to check the satisfiability of $\hat{\varphi}$. If $\hat{\varphi}$ turns out to be unsatisfiable, then so is φ since $\hat{\varphi} \geq \varphi$. On the other hand, if $\hat{\varphi}$ is satisfiable, an ILP solver is called to verify the consistency of that solution. This is done by constructing and solving a system of simultaneous linear integer constraints corresponding to the satisfying assignments found for the indicator variables. If the integer constraints are found to be inconsistent, the abstract formula $\hat{\varphi}$ is *refined* by eliminating its current solution and the SAT solver is re-invoked to find another solution. The procedure terminates if the ILP solver establishes the consistency of one of these SAT solutions or proves that all the (potentially exponential) satisfying assignments to $\hat{\varphi}$ are inconsistent. The combination strategies employed in these classes of solvers are either Shostak-like [29] combination methods or the ones based on work by Nelson-Oppen [24]. For a complete survey of these solvers, the reader is referred to [23].
2. **SAT-Based Finite-Domain Instantiation Methods.** Recently there has been increasing interest in a special class of QFP formulas in which most linear constraints are separation constraints of the form $x - y \leq d$; where x and y are integer variables and d is a constant. In [28], the authors computed an upper bound on integer variables and therefore could set the number of bits in each variable in order to reduce the problem to an equi-satisfiable finite-domain and consequently Boolean problem. They showed that the number of bits per integer variable is linearly proportional to the number of non-separation constraints and logarithmically to the number and size of non-zero coefficients. The separation constraints, on the other hand, are pre-processed using the EIJ method [31] which relies on augmenting the Boolean problem with "transitivity constraints" on the values of indicator variables in order to rule out assignments to those variables that do not have any corresponding solutions in the ILP problem. In the worst case, this process can add an exponential number of transitivity constraints depending on the number of separation constraints.
3. **Automata-Based Methods** [15]**.** The key idea here is to convert the MIB formula to a deterministic finite-state automaton (DFA) such that the language of this automaton corresponds to the set of all solutions of the MIB formula. These methods seem to be efficient for formulas whose integer constraints have large coefficients. However, as observed in [15], the effi-

ciency of this technique declines considerably as the number of variables and/or constraints increases.

4. **Integer Linear Programming Methods.** In these techniques, disjunctions are either removed using Big-M method and the result is checked for satisfiability using a Simplex-based ILP solver [10, 17], or they are enumerated in a Disjunctive Normal Form and then solved using Fourier-Motzkin algorithm [26].

The method we introduce in this paper belongs to the SAT-based abstraction/refinement category. Unlike [4, 20], however, in which the SAT and ILP solvers are loosely integrated, our approach tightly integrates a specialized transitive-closure algorithm with the SAT solver. It achieves its performance by taking advantage of the structure of MIB formulas that arise in verification applications. Specifically, it capitalizes on the fact that the majority of the linear integer constraints involve at most two variables that have unit coefficients, namely UTVPI constraints. Such constraints can be checked quickly using a transitive-closure algorithm which is tightly integrated within the search process of a modern SAT solver. The UTVPI solver can yield implications and generate conflict-induced learned constraints in the SAT solver as soon as such combinations are detected. The non-UTVPI constraints, on the other hand, are handled off-line and may yield conflict-induced UTVPI constraints that help to minimize the number of calls on the non-UTVPI solver. For this purpose, a novel strategy based on theory of cutting-planes is adopted in order to learn from the conflicts detected among the non-UTVPI constraints.

These choices work extremely well in practice since the number of UTVPI constraints is usually much larger than the number of non-UTVPI constraints [11, 28]. Unlike the method of [28, 31] whose efficiency heavily relies on having a low number of separation and non-separation constraints, our method while benefitting from the separation of the UTVPI and non-UTVPI solvers, has no such dependencies on the number of constraints.

The remainder of this paper is organized as follows. In Section 2 we cover some preliminaries. In Section 3 our adopted incremental method in solving UTVPI problems is described. Section 4 covers our approach for encoding and solving the MIB problem. Experimental results are reported in Section 5 and we conclude in Section 6.

2 Preliminaries

An integer linear constraint has the general form

$$\sum_{i=1}^{n} a_i x_i \sim b$$

where $a_i, b, x_i \in \mathbb{Z}$ and $\sim \in \{>, \geq, <, \leq, =, \neq\}$. The special form $a_i x_i + a_j x_j \leq b$ with $a_i, a_j \in \{0, \pm 1\}$ is referred to as a *unit two-variables-per-inequality* (UTVPI) constraint. Single-variable constraints are treated as UTVPI constraints by introducing a dummy variable with a zero coefficient. A constraint

$$
\begin{aligned}
&\varphi(A_1, A_2, A_3, A_4, u, v, w, x, y, z) = \\
&\quad [\, A_1 \vee (u - w \leq 5)\,] \wedge && (\omega_1)\\
&\quad [\, A_2 \vee (v + w \leq 6)\,] \wedge && (\omega_2)\\
&\quad [\, A_3 \vee (z = 0)\,] \wedge && (\omega_3)\\
&\quad [\, A_4 \vee (u + v \geq 12)\,] \wedge && (\omega_4)\\
&\quad [\, \neg A_3 \vee \neg A_4\,] \wedge && (\omega_5)\\
&\quad [(x = z + 1) \vee (x = z + 3) \vee (x = z + 5) \vee (x = z + 7)\,] \wedge && (\omega_6)\\
&\quad [(y = z + 2) \vee (y = z + 4) \vee (y = z + 6)\,] \wedge && (\omega_7)\\
&\quad [(u + v - 4x - 4y = 0)\,] && (\omega_8)
\end{aligned}
$$

Fig. 1. Example MIB-CNF instance

that has more than two variables or that has non-unit coefficients will be referred to as a non-UTVPI constraint. Note that unit two-variable integer equalities and negated-equalities can be transformed to conjunction and disjunction of two UTVPI constraints respectively.

The modern algorithms for propositional SAT are quite well-known and will not be elaborated here; in particular we assume that the reader is familiar with the concepts of Boolean Constraint Propagation (BCP) and its efficient implementation using watched literals, conflict analysis, clause recording, non-chronological backtracking, the VSIDS decision heuristic, restarts, etc. For details on these, readers are referred to [21, 22].

Notationally, we will use A_1 , A_2 , ... (resp. B_1 , B_2 , ...) to denote the original (resp. indicator) Boolean variables in a MIB-CNF formula. Integer variables will be denoted by lower case letters such as u , x , y , ... We will use the formula in Figure 1 as a running example throughout the paper.

3 Deciding Systems of UTVPI Integer Constraints

Problems consisting of conjunctions of UTVPI constraints can be decided using generic ILP solvers such as CPLEX [17] or XPRESS-MP [10]. However, the full-scale Simplex techniques adopted in these solvers do not take advantage of the simple structure of UTVPI constraints and cannot be efficiently integrated within the backtrack search process of modern SAT solvers.

The solution method we adopt in this paper for deciding systems of UTVPI integer linear constraints is a polynomial-time transitive closure algorithm proposed by Jaffar et al. [19] which, in turn, is an extension of Shostak's method for TVPI *real* constraints [30].

A set of UTVPI constraints is said to be *transitively closed* if for each pair of constraints sharing a variable with opposite signs there exists an inequality constraint between the two remaining variables. For instance, the transitive closure of $\{x - y \leq d\,,\, y + z \leq d'\}$ is $\{x - y \leq d\,,\, y + z \leq d'\,,\, x + z \leq d + d'\}$ and we

say that $x + z \leq d + d'$ is *implied by* $x - y \leq d$ and $y + z \leq d'$. In the MIB-CNF context, we will be interested in a dynamically-changing set of UTVPI constraints. To keep such a set transitively closed, whenever a new constraint is added to the set, all its implied constraints must be derived and added. When an implied constraint ends up involving a single variable, it may need to be *tightened* in order to maintain the unit coefficient property. Specifically, the constraint implied by $ax + by \leq d$ and $ax - by \leq d'$ (recall that $a, b \in \{0, \pm 1\}$) is $2ax \leq d + d'$ whose tightening yields $ax \leq \lfloor (d + d')/2 \rfloor$. It is easy to show that the worst-case complexity of maintaining a tightened and transitively-closed set of UTVPI constraints is quadratic in time and space.

To illustrate consider the following set of UTVPI constraints:

$$C = \{y + z \leq 4\ ,\ x - y \leq 5\ ,\ x + y \leq 2\} \tag{1}$$

The transitively-closed and tightened set of constraints derived from (1) is easily shown to be:

$$\begin{aligned} \text{Trans}(C) &= C \cup \{x + z \leq 9\ ,\ 2x \leq 7\} \\ \text{Tighten}(\text{Trans}(C)) &= C \cup \{x + z \leq 9\ ,\ x \leq 3\} \end{aligned} \tag{2}$$

Jaffar et al. [19] showed that a set of UTVPI integer constraints C is satisfiable iff Tighten(Trans(C)) does not contain a constraint of the form $0 \leq d$ where $d < 0$. An example of an unsatisfiable constraint set is:

$$\begin{aligned} C &= \{y - z \leq 1\ ,\ x - y \leq 1\ ,\ z - x \leq -3\} \\ \text{Tighten}(\text{Trans}(C)) &= C \cup \{x - z \leq 2\ ,\ 0 \leq -1\} \end{aligned} \tag{3}$$

Our MIB-CNF SAT solver maintains a database of transitively closed and tightened constraint sets. Specifically, suppose that in the course of searching for a solution the sequence of UTVPI constraint sets C_1, C_2, C_3, ... is generated. As each such set C_i is produced, the corresponding Trans(Tighten(C_i)) set is computed incrementally by adding/removing any implied constraints when a new constraint is added/removed.

4 The MIB-CNF Solver

The architecture of our proposed MIB-CNF solver is shown in Figure 2. It consists of three separate solvers: a modern Boolean SAT solver, an incremental UTVPI integer solver, and a generic ILP (non-UTVPI) solver. The Boolean solver orchestrates the entire process and interacts very closely with the UTVPI solver. Specifically, the Boolean solver views the UTVPI solver as a "super clause" that participates in the implication and conflict analysis steps of the search process. The non-UTVPI solver, on the other hand, is invoked only when the combined Boolean/UTVPI solver returns a satisfying assignment to

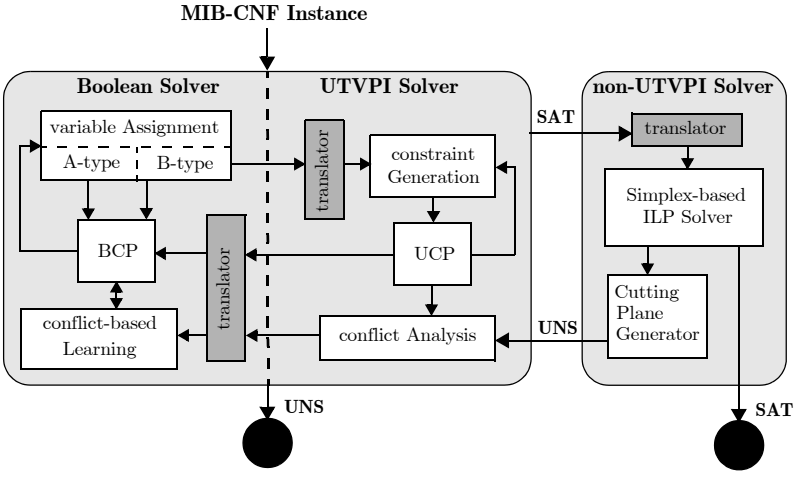

Fig. 2. Overall Architecture of the MIB-CNF Solver

the subset of Boolean and UTVPI constraints. This solver checks the consistency of simultaneous activation of UTVPI and non-UTVPI constraints whose indicator variables are true, by adopting a Simplex-based method. Comparing to other parts of the overall system, the non-UTVPI solver relatively acts as the bottleneck in the process. Therefore, in order to minimize its overall contribution to the run-time of the MIB-CNF solver, two techniques, namely minimizing the size of the ILP problem and learning UTVPI constraints from conflicts in non-UTVPI solver, are adopted. By passing only those absolutely necessary constraints/variables to the non-UTVPI solver, its performance is improved exponentially and by learning from its conflicts in terms of UTVPI constraints, the number of calls to the non-UTVPI solver reduces considerably.

In the remainder of this section we describe how a MIB-CNF instance is encoded for processing by the MIB-CNF solver, and detail the interactions between the Boolean solver and the UTVPI and non-UTVPI solvers.

4.1 Encoding Linear Constraints into the SAT Solver

The first step in solving a MIB-CNF instance is to introduce a set of independent Boolean *indicator* variables B_i to label each of the linear integer constraints and to conjoin the formula with additional relations that establish the equivalence between the indicator variables and their corresponding integer constraints. Specifically, if

$$\varphi = g(A_1,\ldots,A_n,C_1(X),\ldots,C_k(X)) \tag{4}$$

denotes a MIB formula defined over n Boolean variables $A_1, \ldots, A_n$ and k linear integer constraints $C_1(X), \ldots, C_k(X)$, we construct

$\hat{\varphi}(A_1, A_2, A_3, A_4, B_1, \cdots, B_8, u, v, w, x, y, z) =$

$[A_1 \vee B_1] \wedge$ (ω_1)
$[A_2 \vee B_2] \wedge$ (ω_2)
$[A_3 \vee B_3] \wedge$ (ω_3)
$[A_4 \vee B_4] \wedge$ (ω_4)
$[\neg A_3 \vee \neg A_4] \wedge$ (ω_5)
$[B_{61} \vee B_{62} \vee B_{63} \vee B_{64}] \wedge$ (ω_6)
$[B_{71} \vee B_{72} \vee B_{73}] \wedge$ (ω_7)
$[1] \wedge$ (ω_8)

$[B_1 \rightarrow (u - w \leq 5)] \wedge$ (ω_9)
$[B_2 \rightarrow (v + w \leq 6)] \wedge$ (ω_{10})
$[B_3 \rightarrow (z = 0)] \wedge$ (ω_{11})
$[B_4 \rightarrow (u + v \geq 12)] \wedge$ (ω_{12})
$[B_{61} \rightarrow (x = z + 1)] \wedge$ (ω_{13})
$[B_{62} \rightarrow (x = z + 3)] \wedge$ (ω_{14})
$[B_{63} \rightarrow (x = z + 5)] \wedge$ (ω_{15})
$[B_{64} \rightarrow (x = z + 7)] \wedge$ (ω_{16})
$[B_{71} \rightarrow (y = z + 2)] \wedge$ (ω_{17})
$[B_{72} \rightarrow (y = z + 4)] \wedge$ (ω_{18})
$[B_{73} \rightarrow (y = z + 6)] \wedge$ (ω_{19})
$[1 \rightarrow (u + v - 4x - 4y = 0)]$ (ω_{20})

Fig. 3. Encoding of the MIB-CNF instance of Figure 1. Note that constraint ω_8 does not require an indicator variable since it must be satisfied unconditionally

$$\hat{\varphi} = g(A_1, \ldots, A_n, B_1, \ldots, B_k) \wedge \bigwedge_{i=1}^{k} (B_i = C_i) \tag{5}$$

Clearly, $\varphi = \exists(B_1, \cdots, B_k) \cdot \hat{\varphi}$, and the satisfiability/unsatisfiability of $\hat{\varphi}$ implies that of φ. Strict equivalence between the indicator variables and their corresponding linear integer constraints is not necessary, however. The satisfiability/unsatisfiability of φ can be determined by checking the simpler formula

$$\tilde{\varphi} = g(A_1, \ldots, A_n, B_1, \ldots, B_k) \wedge \bigwedge_{i=1}^{k} (B_i \rightarrow C_i) \tag{6}$$

Since $\tilde{\varphi}$ is a superset of $\hat{\varphi}$, this can be viewed as a conservative abstraction: when $\tilde{\varphi}$ is unsatisfiable, so is $\hat{\varphi}$, and when $\tilde{\varphi}$ is satisfiable, it is possible that $\hat{\varphi}$ is unsatisfiable. However, because of the form of $\hat{\varphi}$ and the fact that, by construction, g is positive unate in all indicator variables B_i, the only situation in which this happens is when at least one B_i is 0 and its corresponding linear integer constraint is forced to be satisfied. Such a solution can be changed so that $B_i = 1$, restoring consistency between the integer constraint and its indicator variable without affecting the satisfiability of the original formula.

The practical effect of checking $\tilde{\varphi}$ rather than $\hat{\varphi}$ is that the linear integer solvers need only process those constraints that are "active," i.e. those whose indicator variables are true; constraints whose indicator variables are false can be safely disregarded. This optimization has a major impact on the performance of the MIB-CNF solver.

Figure 3 demonstrates the result of applying this method of encoding on problem of Figure 1.

4.2 The Boolean/UTVPI Solver Interface

The Boolean and UTVPI solvers interact dynamically during the search process and communicate through the indicator variables of the UTVPI constraints (the non-UTVPI constraints are ignored in this phase.) The Boolean solver is responsible for making decisions on the Boolean variables and for performing BCP and conflict analysis. A decision or implication that sets an indi-

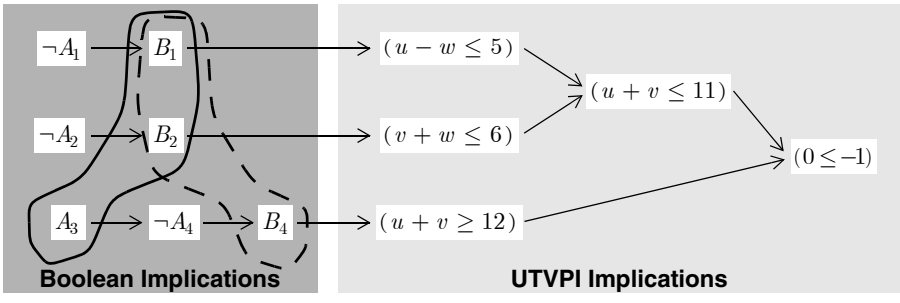

Fig. 4. Implication sequence leading to a conflict in the UTVPI solver. The indicator variables enclosed by the dashed outline are returned by the UTVPI solver as a conflicting assignment. The conflict analysis procedure of the Boolean solver traces back from this assignment to learn and record the conflict-induced clause($\neg A_3 \vee \neg B_1 \vee \neg B_2 \vee$)

cator variable to 1 triggers the UTVPI solver which, in turn, activates the corresponding linear integer constraint and incrementally updates the database of active UTVPI constraints to keep it transitively-closed and tightened. This update can be viewed as continuing the BCP process within the UTVPI solver, and can yield further implications to other indicator variables or can result in conflicts. Inconsistency of the active UTVPI constraints is communicated back to the Boolean solver as a conflicting assignment on the relevant indicator variables. The Boolean conflict analysis procedure takes over at that point to create an appropriate conflict-induced clause [21] and backtracks to eliminate the conflict. The backtrack level is passed to the UTVPI solver so that it can deactivate the linear integer constraints whose indicator variables were reset to 0 or unassigned (as well as any constraints they implied by transitive closure and tightening).

To illustrate the process of implying indicator variables in the UTVPI solver, let B be an indicator variable for a UTVPI constraint C. B is implied to 1 in the UTVPI solver if the process of transitive closure and tightening produces a constraint that is identical or stricter than C. On the other hand, B is implied to 0 if transitive closure and tightening produces a constraint that is inconsistent with C. For example, if B is the indicator variable for $x - y \leq 5$, generating $x - y \leq 3$ causes B to be implied to 1, whereas generating $y - x \leq -6$ causes the implication of B to 0.

The handling of UTVPI conflicts is illustrated in Figure 4 for our running example.

4.3 The Non-UTVPI Solver Theorem 1. Proof:

When the combined Boolean/UTVPI solver returns a satisfying solution, the non-UTVPI solver must be invoked to check the consistency of that solution against any activated non-UTVPI constraints. This can be naively done by collecting *all* of the active linear integer constraints, i.e., the UTVPI and non-UTVPI constraints whose indicator variables are set to 1 in the current solution, and passing them on to a generic Simplex-based ILP solver [10, 17]. As

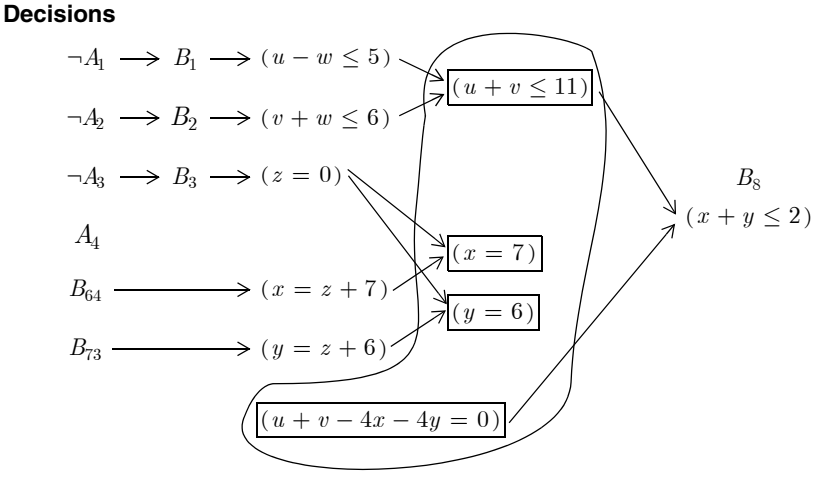

Fig. 5. Conflict analysis in the non-UTVPI solver. After determining that the highlighted constraints are inconsistent, the non-UTVPI solver can either return the conflicting assignment $(B_1 \wedge B_2 \wedge B_3 \wedge B_{64} \wedge B_{73})$, or it can derive and return the learned UTVPI constraint $(x+y \leq 2)$ along with the learned clause $(B_1 \wedge B_2 \wedge \neg B_8)$. The latter is preferable since it minimizes future invocations of the ILP solver

the next theorem shows, however, only a subset of the active UTVPI constraints needs to be processed by the ILP solver.

Theorem 1. A given system of satisfiable transitively-closed UTVPI constraints together with a set of non-UTVPI linear constraints is *equi-satisfiable* with a subset of those UTVPI constraints sharing both variables with variables in the non-UTVPI system and the set of non-UTVPI constraints.

Proof: Chvátal [9] showed that cutting planes provide a canonical way of proving that every integral solution of a given system of linear inequalities satisfies another given inequality. Therefore, a cutting plane proof of unsatisfiability is to prove that there exists a sequence of inequalities such that their non-negative combination results in inequality $0x \leq -1$. Thus, since we know that the

set of UTVPI constraints is satisfiable, the proof of unsatisfiability, if any, should include at least one of the newly added non-UTVPI constraints. In order to yield $0x \leq -1$ from combining linear constraints with those non-UTVPI constraints, it is sufficient to only consider those constraints among the UTVPI constraint that share variables with them knowing that no new constraint can be generated by combining only UTVPI constraints because they are transitively closed. Therefore, the set of non-UTVPI constraints together with those UTVPI constraints which share variables with them is equi-satisfiable with all UTVPI and non-UTVPI constraints. □

To utilize this theorem, the non-UTVPI solver returns the variables associated with its active constraints to the UTVPI solver which uses them to select the subset of UTVPI constraints that must be checked for consistency with the active non-UTVPI constraints. In general, the number of UTVPI constraints that are passed to the ILP solver using this "filter" is much smaller than the total number of active UTVPI constraints, and that directly contributes to much better performance by the ILP solver. Using our running example, the constraints that must be checked by the ILP solver are highlighted in Figure 5.

If the constraints passed to the ILP solver are found to be consistent, the process terminates with a satisfying solution to the original formula. If, on the other hand, the ILP solver finds the constraints to be inconsistent, it must communicate this fact to the Boolean/UTVPI solver which, in turn, must find another solution. This can be done in a manner similar to that used by the UTVPI solver to communicate unsatisfiability to the Boolean solver, namely by returning a set of conflicting indicator variables. This is sufficient, but can be quite inefficient. This is illustrated for our example in Figure 5. After detecting inconsistency, the non-UTVPI solver returns the conflicting assignment $(B_1 \wedge B_2 \wedge B_3 \wedge B_{64} \wedge B_{73})$ to the Boolean solver which uses it to perform conflict analysis and backtracking. Upon finding another satisfying solution, e.g., $(\neg A_1 \wedge \neg A_2 \wedge \neg A_3 \wedge A_4 \wedge B_1 \wedge B_2 \wedge B_3 \wedge B_{63} \wedge B_{73})$, the non-UTVPI solver is invoked again and returns with another conflicting assignment. In fact, this iteration will be repeated twelve times exhausting all possible satisfying assignments to constraints ω_7 and ω_8 before the search process backtracks and reverses an assignment to an A variable.

To reduce the calls to the ILP solver, the non-UTVPI solver can perform its own "intelligent" conflict analysis before returning to the Boolean solver. The goal of this analysis is to derive, using cutting planes, a UTVPI constraint that is implied by the current conflicting assignment. Such a constraint can then be returned to the Boolean/UTVPI solver to help confine the "bactracking" iterations inside that solver. For our example, this is accomplished by combining the non-UTVPI constraint $u + v - 4x - 4y = 0$ with the UTVPI constraint $u + v \leq 11$ yielding $4x + 4y \leq 11$ which is tightened to $x + y \leq 2$. Assigning a fresh indicator variable B_8 to this learned constraint, it is returned to the Boolean/UTVPI solver along with the associated learned clause $(B_1 \wedge B_2 \wedge \neg B_8)$ (see Figure 5). Upon learning this clause, the Boolean/UTVPI solver would eventually know that either A_1 or A_2 should be true by

checking all combinations of B_{6i} and B_{7j} without referring to the non-UTVPI solver and those would be the only satisfying solutions to the problem.

As it is obvious, this process will result in an strictly more powerful learned clause than that of the naive method explained earlier and would considerably help the search algorithm by pruning its infeasible search space more efficiently.

5 Implementation and Experimental Results

The UTVPI solver was implemented within our ARIO SMT solver [3]. ARIO is built on top of MiniSAT [12] and inherits its strategy for random restarts, VSIDS and clause removal. For the Simplex-based ILP solver, we used XPRESS-MP [10] and all experiments were conducted on a Pentium-IV 2800MHz machine with 1 GB of RAM running Linux.

To evaluate our method, we experimented on formulas obtained from the Wisconsin Safety Analyzer (WiSA) project [14]. These instances are concerned with finding API-level exploits by introducing a framework to model low-level

Table 1. WiSA benchmarks

benchmark	Result	Variables int/bin	Constraints		
			CNF	UTVPI	non-UTVPI
s-20-20	SAT	60/973	864	778	6
s-20-30	SAT	60/973	864	778	6
s-20-40	UNS	60/973	864	778	6
s-30-30	SAT	80/1393	1244	1128	6
s-30-40	SAT	80/1393	1244	1128	6
xs-20-20	SAT	82/1116	1046	959	6
xs-20-30	SAT	82/1116	1046	959	6
xs-20-40	UNS	82/1116	1046	959	6
xs-30-40	SAT	112/1606	1516	1399	6

details of API's, and adopting an automatic technique based on bounded, infinite-state model checking. The benchmarks include both satisfiable and unsatisfiable instances and incorporate UTVPI and non-UTVPI constraints combined with Boolean connectors. Un-interpreted functions were initially eliminated using Ackermann's technique [1] and the satisfiability of the resulting formula was tested using different hybrid SAT solvers. Characteristics of the benchmarks used in this evaluation are displayed in Table 1.

The results of running ARIO on the benchmarks of Table 1 are displayed in Table 2. This table also includes CPU run-times obtained from [28] for solving the same instances using the parametrized solution bounds method along with

Table 2. Run-times (in sec.) of ARIO, UCLID and ICS on WiSA benchmarks

benchmark	UCLID time[a]	ICS time	ARIO time		
			UTVPI	non-UTVPI	total
s-20-20	8.78	0.25	0.17	0.01	0.26
s-20-30	9.50	0.37	0.32	0.01	0.61
s-20-40	4.50	286.84	2.77	0.01	5.05
s-30-30	20.89	1.64	0.28	0.01	0.45
s-30-40	19.21	7.41	1.21	0.01	2.06
xs-20-20	26.03	17.77	0.35	0.02	0.57
xs-20-30	21.42	1482.80	0.1	0.01	0.23
xs-20-40	14.18	>3600	173.9	0.01	276.43
xs-30-40	33.22	>3600	1.88	0.06	3.01

[a] from [28] and adjusted to CPU speed of 2.8 GHz

UCLID and also the timings of ICS version 2.0c [13] for the same problems translated by ARIO before applying Ackermann conversion. The hybrid method of ICS is based on lazy, online integration of a non-clausal SAT solver with an incremental, backtrackable constraint engine. The approach adopted in their constraint engine is an extension of Nelson's version of Simplex algorithm where equalities and disequalities are added to a Simplex tableau incrementally. Integer constraints are processed, in the same framework with the addition of cutting planes to preserve their discreteness. For details of their method, the reader is referred to [27].

The clear advantage of ARIO method over the online method of ICS is mainly due to its strategy of separating UTVPI and non-UTVPI constraints, its unique strategy for refinement and more efficient handling of integer constraints.

Comparing to the pre-processing and solution-bound approach of UCLID, our method outperformed on all satisfiable instances due to its more effective learning and on-demand collaboration strategy of its solvers. The cases that UCLID performed better, i.e. mainly small *unsatisfiable* instances, are the ones in which all the transitivity constraints could be efficiently pre-encoded into the SAT solver in the form of CNF clauses, effectively shifting all the search work into the SAT solver rather than sharing it with the slower UTVPI solver as in our method. However, it is important to remember the high dependence of such methods on having a low number of separation and non-separation constraints that cause the solver to slow down exponentially as the number of constraints increase[1].

As demonstrated in Table 3, adopting our intelligent refinement method based on cutting plane theory resulted in a considerable decrease in calls to

[1] We could not obtain the version of UCLID supporting integer non-separation constraints from its developers and therefore our information on its performance is limited to what they reported in [28] and the characteristics of their algorithm.

Table 3. Running ARIO on WiSA benchmarks. Number of runs is the number of times a solution to SAT/UTVPI problem was found to be inconsistent by non-UTVPI engine

benchmark	number of conflicts			Number of runs	
	total	in UTVPI	in Cutting Planes	with Cutting Planes	no Cutting Planes (time)
s-20-20	1111	1057	6	10	84 (0.66 s)
s-20-30	3172	3009	12	8	2066 (15.36 s)
s-20-40	30611	30418	3	1	time-out
s-30-30	1500	1436	2	1	447 (10.17 s)
s-30-40	7631	7281	29	11	273 (7.07 s)
xs-20-20	877	811	11	17	160 (2.09 s)
xs-20-30	396	388	3	1	318 (3.62 s)
xs-20-40	748710	746239	3	1	time-out
xs-30-40	3739	3596	18	16	255 (8.01 s)

non-UTVPI solver and in some cases enormous speed-ups. This table also depicts the number of conflicts due to inconsistencies detected among UTVPI constraints that are all detected online rather than being delayed until the search is complete. Large numbers of such conflicts in these problems makes their online handling vital to the efficiency of the overall algorithm. The number of conflicts detected by cutting planes refers to those conflicts that would have needed a call to the non-UTVPI solver to be detected.

6 Related Work, Conclusions and Future Work

In this paper, we presented a new scalable, cooperative and intelligent method for solving decision problems involving integer linear arithmetic constraints together with Boolean variables. The unique characteristics contributing to the efficiency of our combined method can be summarized as follows:

- Separation of UTVPI and non-UTVPI engines based on the general structure of problems in software and hardware verification applications,
- Online collaborative algorithm for solving UTVPI constraints while the Boolean search proceeds,
- Efficient offline communication between the solvers,
- Intelligent and powerful refinement of inconsistencies by the offline solver.

Tight integration of theory solvers into the SAT solver was also suggested in [16] where any given theory (linear arithmetic for instance) is coupled into the generic DPLL-based propositional solver. Similarly, in our method, the UTVPI constraints are tightly integrated into the SAT solver, but non-UTVPI constraints are handled only after a satisfiable solution to the Boolean/UTVPI

problem is found. Our proposed refinement method for inconsistencies in the SAT solutions leads to a considerable reduction in the number of offline calls and effectively prunes the search space of the SAT solver. By not integrating a full-scale linear arithmetic theory into the SAT solver, our approach maintains both efficiency and completeness of the DPLL-based solver.

The use of a modified implication graph to take into account mathematical deductions was also proposed in [18]. In their method, the DPLL-style SAT solver is combined with a constraint solver based on Fourier-Motzkin elimination. The hybrid conflict analysis scheme for finite-domain integer and Boolean variables, introduced in their work, is comparable to the hybrid UTVPI and Boolean learning method adopted in our method and illustrated in Figure 4.

In [6], a layered decision procedure for the satisfiability of linear arithmetic logic, implemented in MathSAT, is presented. MathSAT is mainly organized in a layered hierarchy of solvers in increasing solving capabilities. Similar to our method, MathSAT also differentiates between Difference Logic (DL) constraints of the form $x - y \sim c$ (a special class of UTVPI constraints) and general linear arithmetic constraints. The former is solved using a Bellman-Ford algorithm and the latter by a simplex-based solver. Integer constraints are handled using branch-and-bound for small problems or Omega Test [26] as the last resort. Unlike MathSAT, in our method the UTVPI constraints are tightly integrated into the SAT solver and by adopting an incremental approach for solving the UTVPI problem and taking advantage of the property introduced in Theorem 1, the integer problem becomes considerably smaller, effectively eliminate the need to the computationally-expensive Omega Test.

Compared to these methods, our method is more robust and reliable as it preserves the completeness of the procedure while maintaining a high level of efficiency. As we learn more about the trade-offs involved, we will be able to develop effective integration strategies that outperform individual techniques.

One promising direction of future research involves improving the efficiency of the collaboration between the SAT solver and its UTVPI engine and also extending the cutting plane techniques described above to add more refinement power in order to minimize the number of inconsistent solutions. Since in most verification applications, the integer solvers are the bottleneck in the SAT-based algorithms, the methods described in this paper when combined with other ILP methods provide a viable option. Specifically, the independent characteristics of our method together with its ability to collaborate with the SAT solver efficiently, makes it more practical for large problems and more amenable to parallelization.

Acknowledgement

This work was funded in part by the National Science Foundation (NSF) under ITR grant No. 0205288. We also would like to thank Vinod Ganapathy for providing us with WiSA benchmarks.

References

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

[2] T. Amon, G. Borriello, T. Hu, and J. Liu, "Symbolic Timing Verification of Timing Diagrams Using Presburger Formulas," *DAC*, pp 226-231, 1997.

[3] ARIO SMT Solver, http://www.eecs.umich.edu/~ario

[4] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowics, and R. Sebastiani, "A SAT-Based Approach for Solving Formulas over Boolean and Linear Mathematical Propositions," *CADE*, pp. 193-208, 2002.

[5] C. Barrett, D. Dill, and J. Levitt , "Validity Checking for Combinations of Theories with Equality," *FMCAD, LNCS 1166*, pp. 187-201, 1996

[6] M. Bozzano, R. Bruttomesso, A. Cimatti, T.A. Junttila, P. van Rossum, S. Schulz, and R. Sebastiani, "An Incremental and Layered Procedure for the Satisfiability of Linear Arithmetic Logic," *TACAS*, pp 317-333, 2005.

[7] R. Brinkmann and R. Drechsler, "RTL-datapath Verification Using Integer Linear Programming," *VLSI Design*, pp 741-746, 2002

[8] J. A. Brzozowski and C.J.H. Seger, "Asynchronous Circuits," *Springer*, 1994.

[9] V. Chvátal, "Edmonds polytopes and a hierarchy of combinatorial problems," Discrete Math. vol. 4 pp. 305-337, 1973.

[10] Dash Inc., XPRESS-MP 15.25.03, http://www.dashoptimization.com.

[11] D.L. Detlefs, G. Nelson, and J.B. Saxe, "Simplify: A Theorem Prover for Program Checking," Tech Report HPL-2003-148, HP Labs, 2003.

[12] N. Eén and N. Sörensson, "An Extensible SAT-solver," SAT, pp. 502-508, 2003.

[13] J.C. Filliatre, S. Owre, H. Rueß and N. Shankar, "ICS: Integrated Canonizer and Solver," *CAV*, pp. 246-249, 2001.

[14] V. Ganapathy, S.A. Seshia, S. Jha, T.W. Reps, R.E. Bryant, "Automatic Discovery of API-Level Exploits," *ICSE*, 2005

[15] V. Ganesh, S. Berezin, and D.L. Dill, "Deciding Presburger Arithmetic by Model Checking and Comparisons with Other Methods," *FMCAD*, pp. 171-186, 2002.

[16] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "DPLL(T): Fast Decision Procedures," *CAV*, pp. 175-188, 2004.

[17] ILOG CPLEX, http://www.ilog.com/products/cplex.

[18] M.K. Iyer, G. Parthasarathy and K.-T. Cheng, "Efficient Conflict-Based Learning in an RTL Circuit Constraint Solver," *DATE*, pp 666-671, 2005.

[19] J. Jaffar, M. Maher, P. Suckey, and R. Yap, "Beyond Finite Domains," *Workshop on Principles and Practice of Constraint Programming*, 1994.

[20] D. Kroening, J. Ouaknine, S. Seshia, and O. Strichman, "Abstraction-based Satisfiability Solving of Presburger Arithmetic," *CAV*, pp.308-320, 2004

[21] J. P. Marques-Silva and K. A. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability," *IEEE Trans. on Computers*, vol. 48(5), pp. 506-521, 1999.

[22] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," DAC, pp. 530-535, 2001.

[23] L. de Moura and H. Rueß, "An Experimental Evaluation of Ground Decision Procedures," *CAV* pp. 162-174, 2004.

[24] G. Nelson, and D.C. Oppen, "Simplification by Cooperating Decision Procedures," ACM Trans. on Programming Languages and Systems vol. 1, pp 245-257, 1979.

[25] M. Presburger, "Uber die Vollstandigkeit eines Gewissen Systems der Arithmetik Ganzer Zahlen, in Welchem die Addition als Einzige Operation Hervortritt," Comptes-rendus du premier congres des mathematiciens des pays slaves, 395: pp 92-101, 1929.

[26] W. Pugh, "The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis," *ACM conf. on Supercomputing*, pp. 4-13, 1991.

[27] H. Rueß, and N. Shankar, "Solving Linear Arithmetic Constraints," SRI International Tech Report CSL-SRI-04-01, January 2004.

[28] S. Seshia and R. Bryant, "Deciding Quantifier-Free Presburger Formulas Using Parameterized Solution Bounds," *LICS*, pp. 100-109, 2004.

[29] R. Shostak, "Deciding Combination of Theories," *Journal of the ACM* vol. 31 pp. 1-12, 1984.

[30] R. Shostak, "Deciding Linear Inequalities by Computing Loop Residues," *Journal of the ACM*, vol. 28(4) pp. 769-779, 1981.

[31] O. Strichman, S.A. Seshia, and R.E. Bryant " Deciding Separation Formulas with SAT", *CAV*, pp 209-222, 2002.

[32] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A First Step Towards Detection of Buffer Overrun Vulnerabilities," *Network and Distributed System Security Symposium*, Internet Society, 2000.

[33] J.M. Wilson, "Compact Normal Forms in Propositional Logic and Integer Programming Formulations," *Computers and Operation Research*, pp. 309-314, 1990.