Abstraction-based Satisfiability Solving of Presburger Arithmetic

Daniel Kroening, Joël Ouaknine
Sanjit Seshia, Ofer Strichman

January 2004
CMU-CS-04-100

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research was sponsored by the Semiconductor Research Corporation (SRC) under contract no. 99-TJ-684, the National Science Foundation (NSF) under grants no. CCR-9803774 and CCR-0121547, the Office of Naval Research (ONR) and the Naval Research Laboratory (NRL) under contract no. N00014-01-1-0796, and the Army Research Office (ARO) under contract no. DAAD19-01-1-0485. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of SRC, NSF, ONR, NRL, ARO, the U.S. Government or any other entity.

Keywords: Presburger arithmetic, Boolean satisfiability, theorem proving, abstraction.

Abstract

We present a new abstraction-based framework for deciding satisfiability of quantifier-free Presburger arithmetic formulas. Given a Presburger formula ϕ, our algorithm invokes a SAT solver to produce proofs of unsatisfiability of approximations of ϕ. These proofs are in turn used to generate abstractions of ϕ as inputs to a theorem prover. The SAT-encodings of the approximations of ϕ are obtained by instantiating the variables of the formula over finite domains. The satisfying integer assignments provided by the theorem prover are then used to selectively increase domain sizes and generate fresh SAT-encodings of ϕ. The efficiency of this approach derives from the ability of SAT solvers to extract small unsatisfiable cores, leading to small abstracted formulas. We present experimental results which suggest that our algorithm is considerably more efficient than directly invoking the theorem prover on the original formula.

1 Introduction

Decision procedures for arithmetic over the integers have many applications in formal verification. For instance, the quantifier-free fragment of Presburger arithmetic has been used in infinite-state model checking [7], symbolic timing verification [2], and RTL-datapath analysis [6]. Unfortunately, the satisfiability problem for quantifier-free Presburger arithmetic is known to be NP-complete [22]. Consequently, efficient techniques and tools for solving such problems are very valuable.

In this paper, we present an abstraction-based algorithm for the satisfiability solving of quantifier-free Presburger formulas (QFP formulas for short). Presburger arithmetic is the first-order theory of linear arithmetic over the non-negative integers. It was shown to be decidable in [24], although the best-known decision algorithms have complexity triply exponential in the size of the formula [26]. For many applications, however, the quantifier-free fragment of Presburger arithmetic suffices.

The algorithm we propose receives as input a QFP formula and attempts to satisfy it over a small range of integers, through a Boolean encoding of the formula and interaction with a SAT solver. In case the Boolean formula is found to be unsatisfiable, the SAT solver is able to supply us with a proof of this fact, in the form of a (small) unsatisfiable core. This in turn can be used to pinpoint the linear arithmetic constraints of the original QFP formula that cannot be satisfied over our bounded domain. This (likewise small) set of linear constraints represents an abstraction of the original QFP formula: whenever these constraints cannot be satisfied over the whole of the non-negative integers, the original formula is unsatisfiable as well. The advantage of this operation is that it is generally much easier to solve the abstracted formula than the original one. We can do so by using any of the existing decision procedures for Presburger arithmetic. The abstracted formula may however be satisfiable; in that case, we increase the size of our bounded integer domain to accommodate the satisfying integer assignment supplied by the decision procedure. We then repeat the whole process until the original formula is shown to be either satisfiable or unsatisfiable.

Note that while the domain over which we work (the integers) is infinite, termination is guaranteed provided the inner Presburger decision procedure we use is itself complete.

Our implementation of this algorithm uses the SAT solvers zChaff [29] and SMVSAT[1], as well as the commercial constraint-solving package CPLEX [15]. Our experimental results, which include both random formulas as well as industrial benchmarks, suggest that our algorithm is considerably more efficient than directly invoking our inner CPLEX-based Presburger decision procedure on its own.

---

[1] We thank Ken McMillan for providing us with this proof-generating SAT solver.

1.1 Related work

There currently exist a number of algorithms and tools for solving QFP, some of which we discuss in Section 2. We refer the reader to the excellent surveys [16, 12] for a more detailed presentation of the matter. While no single technique is found to dominate all others, Ganesh *et al.* [12] report that ILP-based methods perform best in most contexts. To the best of our knowledge, however, no existing tools make use of the sort of abstractions that we have just described.

McMillan and Amla [20] use a related technique to accelerate model checking algorithms over finite Kripke structures. More precisely, they invoke a bounded model checker to decide which state variables should be made visible in order to generate a 'good' abstraction for the next iteration of model checking. Our approach differs from theirs in several respects: we work over an infinite domain, we use a Presburger decision procedure instead of a model checker, and we seek to eliminate constraints rather than variables.

Henzinger *et al.* [13] use a theorem prover to refute counterexamples generated by predicate abstraction in software verification. When the counterexample is shown to be spurious, the proof is 'mined' for new predicates to use in predicate abstraction. Although similar in spirit, this approach differs significantly from ours in both the domain of application and the techniques used.

Our framework also bears certain similarities with automated counterexample-guided abstraction refinement [17]. Chauhan *et al.* [8], for example, use a SAT solver to derive an abstraction sufficient to refute a given abstract counterexample in model checking. The successive abstractions they obtain, however, are cumulative, in that state variables made visible at some point are never subsequently re-hidden. In contrast, our approach generates a fresh abstraction every time.

2 Preliminaries

2.1 Boolean satisfiability

We begin by recalling some well-known facts concerning (propositional) Boolean formulas and Boolean satisfiability.

Let $b_1, b_2, \ldots$ be Boolean variables. A *literal* is either a b_i or its negation. A *(Boolean) clause* is a disjunction of zero or more literals—by convention, the empty clause is equivalent to *False*.

Let ϕ be a Boolean formula with free variables $b_1, \ldots, b_n$. It is possible to manufacture a Boolean formula $\mathsf{cnf}(\phi)$ with free variables $b_1, \ldots, b_{n+p}$ (where the $b_{n+1}, \ldots, b_{n+p}$ are fresh Boolean variables), such that

- $\mathsf{cnf}(\phi)$ is in conjunctive normal form (CNF): $\mathsf{cnf}(\phi) = \bigwedge_{j=1}^{m} B_j$, where each B_j is a Boolean clause,
- $\mathsf{cnf}(\phi)$ is satisfiable iff ϕ is satisfiable; more precisely, $\exists b_{n+1}, \ldots, b_{n+p} \centerdot \mathsf{cnf}(\phi)$ is tautologically equivalent to ϕ, and
- The number of variables and the size of $\mathsf{cnf}(\phi)$ are both linear in the size of ϕ.

Linear-time algorithms for computing $\mathsf{cnf}(\phi)$ are well-known; see, for instance, [23].

A SAT solver is an algorithm which determines, given a Boolean formula ϕ in CNF, whether ϕ is satisfiable. In the affirmative case, the SAT solver will produce a satisfying assignment for ϕ. If, on the other hand, ϕ is unsatisfiable, the SAT solver can be required to produce a *proof of unsatisfiability* [30, 20]. Such a proof in turns yields an *unsatisfiable core*, i.e., an unsatisfiable subset of clauses of ϕ. In practice, SAT solvers tend to generate small unsatisfiable cores. The SAT solvers we have used in our experiments are zChaff [29] and SMVSAT.

2.2 Presburger arithmetic

Presburger arithmetic can be defined as the first-order theory of the structure $\langle \mathbb{N}, 0, 1, \leqslant, + \rangle$, where $\mathbb{N}$ denotes the set of non-negative integers. In this paper, we focus on the *quantifier-free* fragment of Presburger arithmetic.

More precisely, let $x_1, x_2, \ldots$ be variables ranging over non-negative integers. A *linear constraint* is any expression of the form

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

where each a_i and c are integer constants (i.e., $a_i, c \in \mathbb{Z}$), and $\sim$ is a comparison operator ($\sim \in \{<, \leqslant, >, \geqslant, =\}$). *Quantifier-free Presburger formula* (henceforth *QFP formulas*) are Boolean combinations of linear constraints:

Definition 1. *The collection of QFP formulas is defined inductively as follows:*

- *Any linear constraint is a QFP formula, and*
- *If ϕ_1 and ϕ_2 are QFP formulas, then so are $\neg\phi_1$, $\phi_1 \wedge \phi_2$, and $\phi_1 \vee \phi_2$.*

Remark 1. Note that an integer variable x can easily be represented in Presburger arithmetic by two non-negative variables: $x = x_+ - x_-$. It is equally straightforward to encode Boolean variables as equalities of the form $x = 1$, together with constraints of the form $x \leqslant 1$ conjoined at the outermost level.

We are interested in the *satisfaction problem* for QFP formulas: given a QFP formula ϕ, is there an assignment of non-negative integers to the variables of ϕ under which ϕ evaluates to *True*?

Ganesh *et al.* [12] present an excellent survey of decision procedures for satisfiability solving of QFP formulas. The abstraction-based algorithm we describe in Section 3 can be used in conjunction with any such decision procedure; it is however desirable that the procedure also generate satisfying integer assignments. We briefly describe in the next two paragraphs how we use the commercial package CPLEX [15], together with the SAT solver zChaff, to achieve this.[2] Put succinctly, our decision procedure iteratively refines Boolean encodings of the

[2] We also experimented with the tool LP_SOLVE [18] but encountered difficulties with certain formulas, for which LP_SOLVE appears to be unsound.

QFP formula based on satisfying assignments from zChaff that are inconsistent with the linear arithmetic. This technique is known as 'lazy explication of axioms', and was originally proposed by the authors of the tools ICS [14, 10], CVC [9, 4, 5], Math-SAT [19, 3], and Verifun [11].

CPLEX uses integer linear programming techniques (and in particular the simplex algorithm) to decide whether a *conjunction* of linear arithmetic constraints is satisfiable. When such a conjunction is satisfiable, CPLEX also generates a satisfying integer assignment.

Our CPLEX-based QFP solver is implemented as follows. Given a QFP formula ϕ, we first extract a 'Boolean skeleton' ϕ_{bool} from ϕ, by simply replacing each linear constraint in ϕ with a fresh Boolean variable. We then invoke zChaff to determine whether ϕ_{bool} is satisfiable. If it is not, then ϕ cannot possibly be satisfiable either, and we terminate. Otherwise, we take the satisfying assignment (say $\overline{b}$) provided by zChaff and form a corresponding conjunction of linear constraints, which we then submit to CPLEX. If CPLEX is able to find a satisfying assignment for this conjunction of linear constraints, then this assignment also satisfies ϕ, and we are done. Otherwise, we augment the Boolean formula ϕ_{bool} with a 'blocking' clause ruling out the Boolean assignment $\overline{b}$ produced earlier by zChaff. We then repeat the procedure with the new Boolean formula until the satisfiability of ϕ is established or refuted.

In the remainder of this paper, let us refer to our implementation of the above algorithm as 'PresSolver'.

Remark 2. Among other existing QFP solvers, let us mention (i) the OMEGA tool [25], which converts a QFP formula into disjunctive normal form and then applies an extension of the Fourier-Motzkin linear programming algorithm on each disjunct; (ii) the automata-based tool LASH [28]; and (iii) the previously mentioned ILP-based tools LP_SOLVE, ICS, and CVC.

Our abstraction-based framework can be used in conjunction with any of these decision procedures, and indeed we intend to carry out a number of experiments with them in the near future. We have so far mostly worked with the commercial package CPLEX mainly because of its high reliability, completeness over the integers, and efficiency. While ICS and CVC are not at present complete over the integers, and are therefore unsuitable for our purposes, their implementors inform us that they are planning to release complete versions of these tools in the near future.

3 Abstraction-based Presburger Satisfiability Solving

We now present the main contribution of this paper, a SAT-based algorithm that generates increasingly precise *abstractions* of QFP formulas. Our abstractions are obtained by eliminating linear constraints from QFP formulas in a conservative manner. The choice of which constraints to eliminate is guided by an iterative interaction with a SAT solver.

Let ϕ be a QFP formula. If we view the linear constraints occurring in ϕ as atomic propositions, we can convert ϕ into a satisfaction-equivalent QFP formula

ϕ' in CNF by invoking the procedure described in Section 2.1. Note that ϕ' may require the introduction of fresh Boolean variables; as discussed in Remark 1, these are modeled as new constrained integer variables. For the remainder of this section, let us therefore assume without loss of generality that the QFP formula ϕ is given to us in CNF.

Write $\phi = \bigwedge_{j=1}^{m} C_j$, with each C_j a (Presburger) clause. (A Presburger clause is a disjunction of linear constraints.) Let $x_1, \ldots, x_n$ be the collection of variables appearing in ϕ. Suppose we are given a function *size* which assigns to each variable x_i a positive integer $size(x_i)$. Intuitively, $size(x_i)$ denotes the maximum number of bits allowed in the binary representation of x_i; put another way, $size(x_i)$ implicitly represents the constraint $x_i < 2^{size(x_i)}$.

Let C_j^{size} stand for the formula $C_j \wedge \bigwedge_{i \in J_j} x_i < 2^{size(x_i)}$, where J_j is the set of indices of the variables that appear in C_j. We can encode C_j^{size} as an equivalent *Boolean* formula $\mathsf{bool}(C_j^{size})$, as follows. For each variable x_i appearing in C_j, allocate $size(x_i)$ Boolean variables, one for each of the bits allowed in the binary representation of x_i. The linear constraints in C_j are then encoded as Boolean formulas on these Boolean variables. Note that the encoding uses exact (i.e., arbitrary-precision) bit-vector arithmetic.

Next, let $B_j^{size} \mathrel{\widehat{=}} \mathsf{cnf}(\mathsf{bool}(C_j^{size}))$ denote the CNF representation of this Boolean formula, ensuring in the process that all newly introduced auxiliary Boolean variables are fresh. Write

$$\phi^{size} \mathrel{\widehat{=}} \bigwedge_{j=1}^{m} C_j^{size} \qquad \text{and} \qquad \mathsf{cbe}(\phi^{size}) \mathrel{\widehat{=}} \bigwedge_{j=1}^{m} B_j^{size} \quad .$$

(Here cbe stands for 'CNF Boolean Encoding'.) The two main points are:

- The QFP formula ϕ^{size} is satisfiable iff the Boolean formula $\mathsf{cbe}(\phi^{size})$ is satisfiable, and
- Any satisfying assignment for ϕ^{size} is also a satisfying assignment for ϕ.

Observe that since each B_j^{size} is a conjunction of Boolean clauses, $\mathsf{cbe}(\phi^{size})$ is itself in CNF; let us therefore write $\mathsf{cbe}(\phi^{size}) = \bigwedge_{k=1}^{p} A_k$. Note that it is straightforward when building $\mathsf{cbe}(\phi^{size})$ to maintain a table recording, for each clause A_k of $\mathsf{cbe}(\phi^{size})$, its 'origin' $\mathsf{orig}(A_k) = C_j$ in ϕ. While it is possible for several Presburger clauses to yield a common Boolean clause A_k, $\mathsf{orig}(A_k)$ only records one of them. It is clear that, whenever $\mathsf{orig}(A_k) = C_j$, any satisfying integer assignment for C_j^{size} yields a corresponding satisfying Boolean assignment for A_k.

We now come to the crux of the paper, our abstraction-based algorithm for solving QFP formulas, which we call 'ASAP' (Abstraction-based Satisfiability Algorithm for Presburger). ASAP takes as input a QFP formula ϕ in CNF with free variables $x_1, \ldots, x_n$, and either outputs a satisfying assignment of non-negative integers to these variables, or declares ϕ to be unsatisfiable. ASAP repeatedly invokes as subroutines both a SAT solver (SMVSAT) and a quantifier-free Presburger arithmetic decision procedure (PresSolver, which itself is based on CPLEX and zChaff).

ASAP first attempts to satisfy an over-constrained version of ϕ in which the integer variables are only allowed to range over a bounded domain. This is achieved by encoding the over-constrained QFP formula as a Boolean formula, which is then given as input to SMVSAT. If a satisfiable assignment is found, then ϕ is clearly also satisfiable, and an integer witness is easily extracted from the satisfying Boolean assignment produced by SMVSAT. Otherwise, SMVSAT returns an unsatisfiable core, which is in turn used to pinpoint a subset of the clauses of ϕ as unsatisfiable over the chosen bounded domain. The conjunction of these clauses is clearly a conservative abstraction of ϕ, in that if it is unsatisfiable then so is ϕ. We therefore run PresSolver on this abstracted QFP formula. If it is found to be satisfiable, we increase the size of our bounded domain to accommodate the satisfying assignment supplied by PresSolver, and repeat the whole process. We continue until a conclusive judgment on the satisfiability of ϕ is obtained.

Section 4 presents experimental evidence which suggests that ASAP is considerably more efficient than PresSolver on its own. ASAP is described in pseudocode in Figure 1.

Algorithm `ASAP`
Input: `QFP formula` ϕ `in CNF with free variables` $x_1, \ldots, x_n$
Output: `satisfying assignment for` ϕ `or 'UNSAT'`

```
let size(x_i) = 1 for each i
repeat forever
   run SMVSAT on cbe(φ^size)
   if cbe(φ^size) is satisfiable then
      return(corresponding satisfying assignment for φ^size)
   else
      let ⋀_{k∈K} A_k be an unsatisfiable core of cbe(φ^size)
      let ψ = ⋀_{k∈K} orig(A_k)
      run PresSolver on ψ
      if ψ is unsatisfiable then
         return('UNSAT')
      else
         let v_1, ..., v_n be a satisfying assignment for ψ
         let size(x_i) = max(⌈log_2(v_i)⌉, size(x_i)) for each i
end
```

Fig. 1. ASAP: an abstraction-based Presburger satisfiability solving algorithm

Theorem 1. *The algorithm ASAP described in Figure 1 is correct and always terminates.*

Proof. We first examine the issue of correctness. Observe that ψ is less constrained a QFP formula than ϕ, in that it contains only a subset of the clauses of ϕ. Thus if ψ is unsatisfiable, then so is ϕ. On the other hand, suppose that

ASAP terminates with an assignment $\overline{v}$ of non-negative integers to the variables. $\overline{v}$ is a satisfying assignment for ϕ^{size}, for some instance of *size*. But since any satisfying assignment for ϕ^{size} is automatically a satisfying assignment for ϕ, $\overline{v}$ is indeed a satisfying assignment for ϕ, as required.

We now claim that, in any execution of ASAP, we never see two identical instances of the QFP formula ψ. Since ψ is always a conjunction of a subset of the clauses of ϕ, it only has finitely many possible instantiations, which immediately entails the termination of the algorithm.

It remains to establish our claim. Suppose, on the contrary, that two identical instances of ψ are observed in a given execution of ASAP. The first time around, an unsatisfiable core $\bigwedge_{k \in K} A_k$ of $\mathsf{cbe}(\phi^{size})$ is obtained, and the function *size* is then subsequently increased to $size'$ to accommodate a satisfying integer assignment $\overline{v}$ for ψ. In other words, $\psi = \bigwedge_{k \in K} \mathsf{orig}(A_k)$ and $\psi^{size'}[\overline{v}]$ evaluates to *True*. (Here $\psi^{size'}$ denotes the QFP formula ψ conjoined with linear constraints of the form $x_i < 2^{size'}$.)

Some iterations later, we encounter a second unsatisfiable core $\bigwedge_{l \in L} A_l$ of $\mathsf{cbe}(\phi^{size''})$ such that $\psi = \bigwedge_{l \in L} \mathsf{orig}(A_l)$. Writing $C_{j(l)} \widehat{=} \mathsf{orig}(A_l)$, we have that $C_{j(l)}^{size'}[\overline{v}]$ evaluates to true for every $l \in L$, since $\overline{v}$ is a satisfying assignment for $\psi^{size'}$. Since each iteration of the `repeat` loop increases (pointwise) the function *size*, we conclude that $size'' \geqslant size'$, and therefore that $C_{j(l)}^{size''}[\overline{v}]$ also evaluates to true for every $l \in L$. Let $\overline{b}$ be the Boolean assignment to the bit-variables prescribed by $size''$ corresponding to the integer assignment $\overline{v}$. We immediately get that $A_l[\overline{b}]$ evaluates to *True* for each $l \in L$, and therefore that $\bigwedge_{l \in L} A_l$ is satisfiable, contradicting our earlier hypothesis. □

Remark 3. We record the following observations concerning ASAP:

- Any ILP-based solver, such as PresSolver, offers the option of generating satisfying integer assignments that moreover minimize some linear 'objective function' $f(x_1, \ldots, x_n)$. In the case at hand, it is desirable that satisfying assignments be as compact as possible; more precisely, they should ideally minimize the number of new bits that are required for their representation. A simple linear function which approximates this requirement is $f(x_1, \ldots, x_n) \widehat{=} \sum_{i=1}^{n} x_i$, and in fact that is the function that PresSolver uses. Note that while minimizing the number of bits subsequently leads to easier queries for the SAT solver, the minimization requirement is an additional burden for the ILP-based Presburger solver.
- Note that, while our inner Presburger decision procedure PresSolver generates satisfying integer assignments, many theorem proving tools do not. Nonetheless, we could still use a pure decision procedure in ASAP by simply requiring, on every iteration, that the function *size* be increased by 1 for each of its arguments.

4 Implementation and Experimental Results

We implemented our tool ASAP within the UCLID verification system [27], which is implemented in Moscow ML [21], a dialect of Standard ML. In implementing ASAP, we used PresSolver as a decision procedure for QFP formulas, and SMVSAT as a proof-generating SAT solver. SMVSAT outputs a proof as a set of resolution steps. The set of all original (i.e., not introduced by resolution) clauses that appear in this proof constitute the unsatisfiable core. ASAP interacts with PresSolver and SMVSAT using a file-based interface. The total running time for ASAP is the cumulative time spent in generating input for SMVSAT and PresSolver, in running SMVSAT and PresSolver, and in analyzing their output.

We performed an experimental evaluation to investigate whether using PresSolver within ASAP could achieve a significant speed-up over directly using PresSolver on the input formula. We used two benchmark sets of QFP formulas in CNF: randomly generated formulas, and formulas generated in real-world software verification problems.

The experiments were performed on a Linux workstation with an AMD Athlon 1.5 GHz dual-processor CPU, with 3 GB of RAM. Both ASAP and PresSolver are single-threaded.

4.1 Results on Random Benchmarks

We ran both ASAP and PresSolver on a set of 45 randomly generated formulas with a timeout of 1200 seconds. The formulas included both unsatisfiable and satisfiable instances. We generated the formulas recursively as follows: for each node, we randomly select either a boolean operator ($\wedge$, $\vee$, $\neg$) or a relational operator ($=$, $<$, etc.). In case of a relation, we generate a linear constraint, randomly selecting the coefficients of the variables and the constant term from the range $[0, 100]$. The number of variables is fixed, but the number of linear constraints can vary, allowing us to generate over-constrained formulas that have a reasonable likelihood of being unsatisfiable. The depth of nesting of Boolean operators in the formula is bounded, eventually forcing the selection of a relational operator.

Figure 2 compares, for each formula, the total run-time of ASAP with the run-time of PresSolver. In the plot, the x-coordinate of each point is the time taken by ASAP, and the y-coordinate is the time taken by PresSolver. We also plot the diagonal line $y = x$: points above the diagonal correspond to benchmarks on which ASAP outperforms PresSolver, while points below it correspond to benchmarks on which ASAP is outperformed.

The results show that ASAP outperforms PresSolver on most of the benchmarks, completing on all benchmarks within a minute while PresSolver times out on 6 benchmarks. On larger benchmarks for which PresSolver terminates, we notice that ASAP performs an order of magnitude better; the speed-up is more than a factor of 100 on some benchmarks. PresSolver outperforms ASAP on some smaller formulas, but ASAP completes within 4 seconds on all of these;

the reason for PresSolver's superior performance on these is simply because the original formulas themselves are fairly small (about 80 clauses), so that ASAP's extra overhead is comparatively more costly.

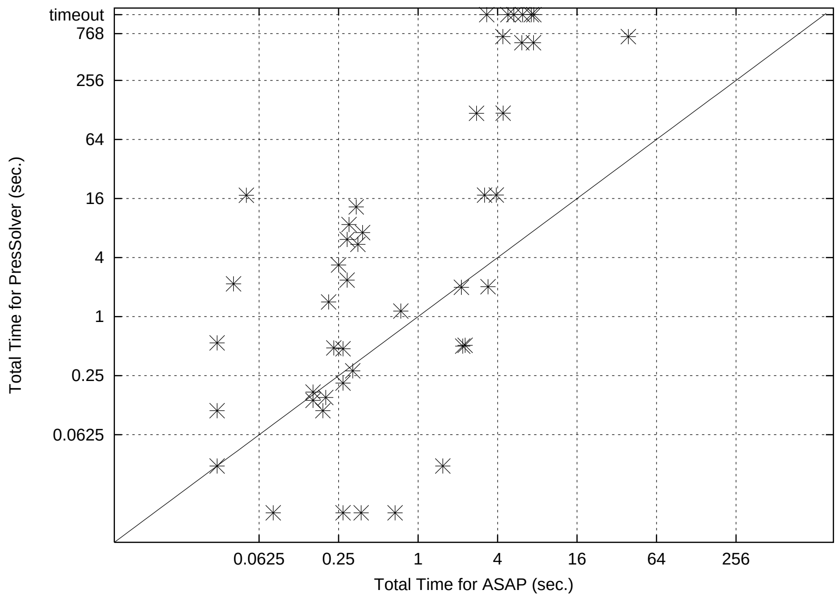

Fig. 2. Comparing ASAP against PresSolver on random benchmarks. The timeout was 1200 seconds. Note the log scale on both axes.

We also investigated how the maximum size of any abstracted formula ψ, measured in terms of number of CNF clauses, compares with that of the original formula ϕ. This was done by computing the ratio of the number of clauses in ψ with the number of clauses in ϕ. We found the smallest such ratio to be 0.009, where the original formulas has 648 clauses and the largest abstraction contains just 6. The largest ratio was 0.206, with 354 clauses in the original formula, and 73 in the largest abstraction. This reduction is the main reason for the speed-ups achieved over directly using PresSolver on ϕ.

4.2 Results on Software Verification Benchmarks

We now report on the second set of experiments performed on formulas generated from software verification. We used a suite of formulas generated in the WiSA project[3] in checking for format string vulnerabilities. The benchmarks include 20 formulas, both satisfiable and unsatisfiable, in an extension of QFP with

[3] `http://www.cs.wisc.edu/wisa`

uninterpreted functions. Uninterpreted functions were first eliminated using Ackermann's technique [1], and both ASAP and PresSolver were run on the resulting QFP formula. Each generated QFP formula is an arbitrary Boolean combination of linear constraints. The number of variables in the formulas ranges from 33 to 43, the number of linear constraints ranges between 197 and 267, and the total number of clauses ranges between 695 and 1054.

PresSolver was unable to solve any of these formulas within a timeout of one hour. On the other hand, ASAP was able to complete on all but one benchmark within the timeout. Table 1 shows the results of running ASAP on the WiSA benchmarks. We give the total time taken by ASAP and the time taken by PresSolver on abstractions generated by ASAP, summed over all invocations of PresSolver. We notice that on all benchmarks, the time taken by PresSolver is the bottleneck for ASAP, accounting for over 90% of the total time on most benchmarks. ASAP takes on the order of a few minutes to solve each formula. SMVSAT took less than 5 seconds on each occasion, with the remaining time spent in generating encodings and parsing output from SMVSAT and PresSolver.

The abstractions generated by ASAP were fairly compact. The last column in Table 1 shows the ratio of the number of clauses in the largest abstraction ψ generated by ASAP to the number of clauses in the original formula ϕ. We see that this ratio is roughly between 10 and 15 percent. Again, the compactness of the abstraction is the main reason why ASAP is able to solve these formulas, while PresSolver is unable to complete on any. Finally, we note that the number of iterations taken by ASAP ranged between 19 and 28.

5 Conclusion

We have presented a novel abstraction-based approach to the satisfiability solving of quantifier-free Presburger formulas. Our experimental results, over both random formulas as well as industrial benchmarks, indicate that embedding a theorem prover for QFP formulas within our framework can achieve significant speed-ups over directly using the prover on the input formula.

In the future, we would like to experiment with a number of other Presburger solvers, and in particular ICS and CVC. While these tools are at present incomplete over the integers, and hence not suitable for our purposes, their implementors inform us that new releases will remedy this, something we look forward to.

Another research direction would be to investigate whether the abstraction-based methodology presented here could be applied to other logics and theories, possibly over different domains such as bit vectors, real numbers, etc.

Acknowledgments

We thank Vinod Ganapathy and Somesh Jha for providing us with benchmark formulas.

Benchmark	ASAP Time		Max. Ratio
	PresSolver Time (sec.)	Total Time (sec.)	$\frac{\#(\psi \text{ clauses})}{\#(\phi \text{ clauses})}$
s-5-3	199.70	230.44	0.139
s-5-5	239.48	262.00	0.142
s-5-6	193.53	215.56	0.148
s-5-7	274.08	303.45	0.142
s-6-5	286.88	310.05	0.125
s-6-6	261.93	292.05	0.132
xs-5-2	408.56	434.16	0.120
xs-5-3	596.82	631.77	0.122
xs-5-5	970.01	994.11	0.124
xs-5-6	934.18	959.76	0.126
xs-5-7	978.83	1008.53	0.124
xs-5-9	126.44	150.90	0.155
xs-5-10	127.93	155.01	0.155
xs-5-11	1034.01	1062.88	0.161
xs-6-2	648.44	683.79	0.103
xs-6-5	1051.57	1116.78	0.106
xs-6-6	1008.85	1043.09	0.110
xs-6-9	132.04	158.97	0.135
xs-6-10	184.51	220.18	0.135
xs-6-11	*	*	–

Table 1. ASAP results on WiSA benchmarks. A "*" indicates that ASAP timed out after 1 hour.

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. In *Proceedings of DAC 97*, pages 226–231, 1997.

[3] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani. A SAT based approach for solving formulas over boolean and linear mathematical propositions. In *Proceedings of CADE 02*, pages 195–210, 2002.

[4] C. Barrett, D. Dill, and A. Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In *Proceedings of CAV 02*, volume 2404, pages 236–249. Springer LNCS, 2002.

[5] S. Berezin, V. Ganesh, and D. L. Dill. An online proof-producing decision procedure for mixed-integer linear arithmetic. In *Proceedings of TACAS 03*, volume 2619, pages 521–536. Springer LNCS, 2003.

[6] R. Brinkmann and R. Drechsler. RTL-datapath verification using integer linear programming. In *Proceedings of VLSI Design*, pages 741–746, 2002.

[7] T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using Presburger arithmetic. In *Proceedings of CAV 97*, volume 1254, pages 400–411. Springer LNCS, 1997.

[8] P. Chauhan, E. M. Clarke, J. H. Kukula, S. Sapra, H. Veith, and D. Wang. Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis. In *FMCAD 02*, pages 33–51, 2002.

[9] CVC. `http://verify.stanford.edu/CVC/`.

[10] L. de Moura, H. Rueß, and M. Sorea. Lazy theorem proving for bounded model checking over infinite domains. In *Proceedings of CADE 02*, pages 438–455, 2002.

[11] C. Flanagan, R. Joshi, X. Ou, and J. B. Saxe. Theorem proving using lazy proof explication. In *Proceedings of CAV 03*, volume 2725, pages 355–367, 2003.

[12] V. Ganesh, S. Berezin, and D. L. Dill. Deciding Presburger arithmetic by model checking and comparisons with other methods. In *Proceedings of FMCAD 02*, volume 2517, pages 171–186. Springer LNCS, 2002.

[13] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of POPL 02*, pages 58–70. ACM, 2002.

[14] ICS. `http://www.icansolve.com`.

[15] ILOG CPLEX. `http://www.ilog.com/products/cplex/`.

[16] P. Janičić, I. Green, and A. Bundy. A comparison of decision procedures in Presburger arithmetic. Research paper no. 872, Division of Informatics, 1997. University of Edinburgh.

[17] R. Kurshan. *Computer-Aided Verification of Coordinating Processes.* Princeton University Press, 1994.

[18] LP_SOLVE. `http://www.freshports.org/math/lp_solve/`.

[19] Math-SAT. `http://dit.unitn.it/~rseba/Mathsat.html`.

[20] K. McMillan and N. Amla. Automatic abstraction without counterexamples. In *Proceedings of TACAS 03*, volume 2619, pages 2–17. Springer LNCS, 2003.

[21] Moscow ML. `http://www.dina.dk/~sestoft/mosml.html`.

[22] C. H. Papadimitriou. On the complexity of integer programming. *Journal of the ACM*, 28(4):765–768, 1981.

[23] D. A. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986.

[24] M. Preßburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. *Comptes-rendus du premier congrès des mathématiciens des pays slaves*, 395:92–101, 1929.

[25] W. Pugh. The Omega Test: A fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, 1991.

[26] R. E. Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the ACM*, 26(2):351–360, 1979.

[27] UCLID. `http://www.cs.cmu.edu/~uclid`.

[28] P. Wolper and B. Boigelot. An automata-theoretic approach to Presburger arithmetic constraints. In *Proceedings of SAS 95*, volume 983, pages 21–32. Springer LNCS, 1995.

[29] zChaff. `http://www.ee.princeton.edu/~chaff/zchaff.php`.

[30] L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formulas. In *Proceedings of SAT 03*, 2003.