Anatomy of Alternating Quantifier Satisfiability (Work in progress)

Anh-Dung Phan[1], Nikolaj Bjørner[2] and David Monniaux[3]

[1] Technical University of Denmark
[2] Microsoft Research
[3] Verimag

Abstract

We report on work in progress to generalize an algorithm recently introduced in [10] for checking satisfiability of formulas with quantifier alternation. The algorithm uses two auxiliary procedures: a procedure for producing a candidate formula for quantifier elimination and a procedure for eliminating or partially eliminating quantifiers. We also apply the algorithm for Presburger Arithmetic formulas and evaluate it on formulas from a model checker for Duration Calculus [8]. We report on experiments on different variants of the auxiliary procedures. So far, there is an edge to applying SMT-TEST proposed in [10], while we found that a simpler approach which just eliminates quantified variables per round is almost as good. Both approaches offer drastic improvements to applying default quantifier elimination.

1 Introduction

Can formulas with nested quantifiers be checked effectively for satisfiability? Several algorithms exist in the context of Quantified Boolean Formulas that handle alternation of quantifiers [12, 1]. They are specialized for eliminating variables over Booleans. An algorithm for alternating quantifier satisfiability was given in [10] for the case of linear arithmetic over the Reals. It integrates tightly an *All-SMT* loop and projection based on Fourier-Motzkin elimination or Chernikov projection. A question arises whether the ideas lift to other projection procedures. Also, are there reasonable alternatives to *All-SMT* and how do they compare? This ongoing work presents a generalized algorithm of that presented in [10] which abstracts the auxiliary procedures. We instantiate the generalization to projection functions based on virtual substitutions, i.e. substitution methods that replace quantifiers by disjunctions of bounded variables. The specialization is for Linear Integer Arithmetic based on Cooper's procedure and used for formulas that arise from Duration Calculus Model Checker (DCMC).

Linear Integer Arithmetic (LIA) or Presburger Arithmetic, introduced by Mojzaesz Presburger in 1929, is a first-order theory of integer which accepts + as its only operation. A classic example of representing some amount of money by 3-cent coins and 5-cent coins appears in LIA as follows:

$$\forall \mathtt{z}\ (\mathtt{z} \geq 8 \rightarrow \exists \mathtt{x}\ \exists \mathtt{y}\ (3\mathtt{x} + 5\mathtt{y} = \mathtt{z}))$$

After Presburger proved decidability of LIA [14], LIA attracted a lot of attention due to applications in different areas. Cooper's algorithm [4] is a *substitution-based* decision procedure. The Omega Test is a *projection-based* decision procedure for LIA and employed in dependence analysis of compilers [13]. A variant that integrates elements from both Cooper's method and the Omega Test is implemented in Z3 [2]. While Z3 can handle non-trivial LIA problems, applications from DCMC also expose limitations of using quantifier elimination alone. The time complexity of all procedures for Presburger Arithmetic is high. Let n denote the length of a LIA formula; running time of any decision procedure is at least $2^{2^{cn}}$ for some constant $c > 0$ [6]. Moreover, Oppen proved a triply exponential upper bound $2^{2^{2^{cn}}}$ for worst-case running time of Cooper's algorithm [11].

This paper is organized as follows. Section 2 presents the generalized algorithm with a few supporting procedures. Section 3 presents different methods for producing candidate formulas. Section 4 instantiates the algorithm with a concrete procedure for virtual substitutions. We discuss implementation details and benchmarks in Section 5 and Section 6 concludes the paper.

2 Alternating Quantifier Satisfiability

This section develops an algorithm that is an abstraction of the alternating quantifier satisfiability algorithm presented for Linear Real Arithmetic in [10]. The abstraction is formulated such that various quantifier elimination methods can be plugged in, including virtual substitutions.

2.1 Definitions

The algorithm being developed relies on two procedures for *extrapolation* and *projection.* We first describe the requirements for these procedures and discuss the main algorithm later.

Definition 1 (Extrapolant)**.** *Given two formulas A and B, a formula C is an extrapolant of A and B if the following conditions are satisfied:*

$A \wedge B$ is `unsat` *then* $C = false$

$A \wedge B$ is `sat` *then* $A \wedge C$ *is* `sat`, $\neg B \wedge C$ *is* `unsat`

We use $\langle A,\ B\rangle$ to denote an extrapolant.

Extrapolation is typically understood as finding new data points outside a set of existing points. Intuitively, C has empty intersection with $\neg B$ and non-empty intersection with A. There are many possible extrapolants for each pair of formulas, and the definition here does not specify how to compute an extrapolant. An example of a trivial extrapolant is described in the following procedure: when $A \wedge B$ is satisfiable we can take B as an extrapolant, otherwise take *false*.

Definition 2 (Projection $\pi x.(C|M)$)**.** *Let M and C be quantifier-free formulas where variable x only occurs in C ($x \notin FV(M)$ where $FV(M)$ denotes the set of free variables in M). Assume $C \wedge M$ is satisfiable. A projection procedure $\pi x.(C|M)$ computes a quantifier-free formula satisfying the conditions:*

1. $FV(\pi x.(C|M)) \subseteq FV(C) \setminus \{x\}$
2. $\pi x.(C|M)$ *is* `sat`
3. $(M \wedge \pi x.(C|M)) \rightarrow \exists x\ C$

Similar to extrapolation we only gave the conditions that projection functions have to satisfy for the developing algorithm to be sound. There is a choice of algorithms for implementing $\pi x.(C|M)$. A possible way is to use *virtual substitutions*, where we derive a quantifier-free formulas by substituting variables by one or more disjunctions. *Virtual substitutions* will be presented with more details in Section 4.

Algorithm 1: $QT(i,\ C,\ \overline{x},\ \overline{M})$

```
if $C \wedge \overline{M}_i$ is unsat then
    return $(false,\ \overline{M})$
end
if $i\ =\ n$ then
    return $(\langle C,\ \overline{M}_i\rangle,\ \overline{M})$
end
$(C',\ \overline{M'}) \leftarrow$
$QT\ (i+1,\ \langle C,\ \overline{M}_i\rangle,\ \overline{x},\ \overline{M})$;
if $C'\ =\ false$ then
    return $(\langle C,\ \overline{M}_i\rangle,\ \overline{M'})$
end
$\overline{M''}_k\ \leftarrow\ \overline{M'}_k,\ \forall\ k \neq i$;
$\overline{M''}_i\ \leftarrow\ \overline{M'}_i \wedge\ \neg(\pi\overline{x}_i.(C'|\overline{M'}_i))$;
return $QT\ (i,\ C,\ \overline{x},\ \overline{M''})$;
```

Algorithm 2: $QE(\overline{x},\ \overline{F}_n)$

```
$\overline{M}_k \leftarrow true,\ \forall\ k < n$;
$\overline{M}_n \leftarrow \overline{F}_n$;
$C \leftarrow false$;
$C' \leftarrow true$;
while $C' \neq false$ do
    $(C',\ \overline{M}) \leftarrow QT(1,\ \neg C,\ \overline{x},\ \overline{M})$;
    $C \leftarrow C \vee C'$;
end
return $C$;
```

2.2 Quantifier Test: algorithm *QT*

QT is defined as a recursive function in Algorithm 1. The four arguments of QT can be explained in the following way: C is a context formula which reflects collected information at current iteration; $\overline{x}$ and $\overline{M}$ are vectors of quantified variables and formulas respectively, and i is the index to access current elements in the above vectors. We use the notation $\overline{M}_i$ to refer to the i-th element of vector $\overline{M}$ and imply the same notation for other vectors.

Given any nested quantified formula $\overline{F}_1$ in the form of $Q_1x'_1Q_2x'_2...Q_kx'_k\ F'$ where $Q_i \in \{\forall,\ \exists\}$ and F' is a quantifier-free formula, we can convert $\overline{F}_1$ into a form of $\forall\overline{x}_1\neg\forall\overline{x}_2...\neg\forall\overline{x}_n\neg\overline{F}_n$ where $\overline{F}_n$ is also quantifier-free. This leads to a sequence of formulas $\overline{F}_i$ such that: $\overline{F}_i \equiv \forall\overline{x}_i\neg\overline{F}_{i+1}$ for $i < n$. Before QT is called, the initial values of $\overline{M}_i$ are initialized to *true* for $i < n$ and $\overline{M}_n$ is initialized to $\overline{F}_n$, The final value of $\overline{M}_1$ is *false* if and only if $\overline{F}_1$ is unsatisfiable.

Theorem 1 (Partial Correctness)**.** *Assume C is satisfiable. The algorithm $QT(i,\ C,\ \overline{x},\ \overline{M})$ returns a pair of the form $(C',\ \overline{M'})$ where C' is an extrapolant of $\langle C,\ \overline{F}_i\rangle$.*

We do not provide a detailed proof, but we briefly discuss correctness and the invariant $\overline{F}_i \Rrightarrow \overline{M}_i$, that are mutually inductive. The main idea is that $\overline{F}_i \Rrightarrow \overline{M}_i$ holds for the initialization step for QT, and $\overline{F}_i \Rightarrow \neg(\pi\overline{x}_i.(C'|\overline{M}_i))$ also holds for each QT iteration; therefore, we strengthen $\overline{M}_i$ in the end of QT and preserve the invariant at the same time. The *if* branches of QT return $\langle C,\ \overline{M}_i\rangle$ when $\overline{M}_i$ cannot be strengthened any more. Moreover, QT also ensures that $\langle C,\ \overline{M}_i\rangle \wedge \overline{F}_i$ is `unsat`. Therefore, the last $\langle C,\ \overline{M}_i\rangle$ (corresponding to the strongest version of $\overline{M}_i$) is also an extrapolant of $\langle C,\ \overline{F}_i\rangle$.

Termination: Algorithm QT does not terminate for arbitrary instantiations of projection and extrapolation. The projection and extrapolation operators we examine here are well-behaved (with respect to termination) in the following sense: (1) The extrapolation procedures do not introduce new atoms, so there will be only a finite number of new extrapolants one can make. (2) The projection procedures are also finitary: they produce only a finite set of projections for the case of linear arithmetic.

2.3 Quantifier Elimination: algorithm QE

A quantifier-free version of $\overline{F}_1$ is obtained by executing QT until saturation in Algorithm 2. The algorithm initializes a vector of formulas $\overline{M}$ and strengthens these formulas as much as possible in a loop.
The intuition of QE is described as follows:

- Run $QT(1,\ \neg false,\ \overline{x},\ \overline{M})$, we obtain a formula C_1 where $C_1 \Rrightarrow \overline{F}_1$.
- Execute $QT(1,\ \neg C_1,\ \overline{x},\ \overline{M})$, we get C_2, a disjoint formula of C_1, where $C_2 \Rrightarrow \overline{F}_1$.
- Run $QT(1,\ \neg(C_1 \vee C_2),\ \overline{x},\ \overline{M})$, we obtain a next formula C_3 where $C_3 \Rrightarrow \overline{F}_1$.
- When $QT(1, \neg C,\ \overline{x},\ \overline{M})$ returns `false`, we get C as a disjunction of disjoint formulas where $C \equiv C_1 \vee C_2 \vee ... \vee C_k$ and $C \Rrightarrow \overline{F}_1$.

2.4 Algorithm QT by example

We use a small example to illustrate the algorithm QT:

$$\forall y\ \exists z\ (z \geq 0 \wedge ((x \geq 0 \wedge y \geq 0) \vee -y - z + 1 \geq 0)) \tag{1}$$

The formulas corresponding to (1) are:

$$\overline{F}_1 = \forall y\ \neg \overline{F}_2, \quad \overline{F}_2 = \forall z\ \neg F_3, \quad \overline{F}_3 = z \geq 0 \wedge ((x \geq 0 \wedge y \geq 0) \vee -y - z + 1 \geq 0)$$

and quantifiers are:

$$\overline{x}_1 = y, \quad \overline{x}_2 = z$$

Algorithm QT also maintains formulas $\overline{M}_1, \overline{M}_2$ and $\overline{M}_3$ that are initialized as follows:

$$\overline{M}_1 = \mathit{true}, \quad \overline{M}_2 = \mathit{true}, \quad \overline{M}_3 = \overline{F}_3 = z \geq 0 \wedge ((x \geq 0 \wedge y \geq 0) \vee -y - z + 1 \geq 0)$$

It maintains the invariants:

$$\overline{F}_i \Rrightarrow \overline{M}_i, \quad FV(\overline{M}_i) \subseteq \{\overline{x}_1, \ldots, \overline{x}_{i-1}\} \quad \text{for } i = 1, 2, 3 \tag{2}$$

Finally, the algorithm propagates a context formula C_i between levels. The context formula is updated during propagation. When C_i is propagated from level i to $i+1$ or $i-1$, it results in a formula (C_{i+1} or C_{i-1}) that has non-empty intersection with C_i and is implied by $\overline{M}_i$. This formula is an *extrapolant* of C_i and $\overline{M}_i$ as defined in Definition 1. The extrapolant C_i on level i satisfies:

$$FV(C_i) \subseteq \{\overline{x}_1, \ldots, \overline{x}_{i-1}\} \tag{3}$$

It contains only variables that are free above level i.

Let us run QT on the sample formula. In the initial state, vector $\overline{M} = \langle \mathit{true}, \mathit{true}, \overline{F}_3 \rangle$, $C_1 = \mathit{true},\ i = 1$.

1. $C_1 \wedge \overline{M}_1$ is $\mathit{true} \wedge \mathit{true}$. It is satisfiable, so let us choose an extrapolant C_2 for C_1 and $\overline{M}_1$. $C_2 := \mathit{true}$ is an extrapolant because $C_1 \wedge C_1$ is satisfiable and $\neg \overline{M}_1 \wedge C_2$ is unsatisfiable. Set $i := 2$.

2. $C_2 \wedge \overline{M}_2$ is also satisfiable and similarly we set $C_3 := \mathit{true},\ i := 3$.

3. $C_3 \wedge \overline{M}_3$, which is $C_3 \wedge \overline{F}_3$, is satisfiable as well. In this case we will propagate back to level 2 a formula that intersects with C_3 and implies $\overline{F}_3$. So let us return $C_3 := z \geq 0 \wedge x \geq 0 \wedge y \geq 0$ and update the level $i := 2$.

4. Now $\overline{F}_2 \equiv \forall z.\neg\overline{F}_3$ implies that $\forall z.\neg C_3 \equiv \neg\exists z.C_3$. So we can strengthen $\overline{M}_2$ with the negation of any formula that implies $\exists z.C_3$. This is a *projection* as denoted by the notation $\pi z.(C_3|\overline{M}_2)$ in Definition 2. It can take the current state of $\overline{M}_2$ into account. In this case we set $\pi z.(C_3|\overline{M}_2)$ to $x \geq 0 \wedge y \geq 0$ and update $\overline{M}_2 := \neg(x \geq 0 \wedge y \geq 0)$.

5. $C_2 \wedge \overline{M}_2$, which is $\neg(x \geq 0 \wedge y \geq 0)$, remains satisfiable. Let us set $C_3 := \neg(y \geq 0),\ i := 3$. C_3 satisfies the conditions for being an extrapolant.

6. $C_3 \wedge \overline{M}_3$ $(= C_3 \wedge \overline{F}_3)$ is still satisfiable. We return the extrapolant $C_3 := z \geq 0 \wedge -y-z+1 \geq 0$ and set $i := 2$.

7. The formula $y \leq 1$ implies $\exists z.C_3$ (they are actually equivalent), so we can update $\overline{M}_2 := \overline{M}_2 \wedge \neg(y \leq 1)$ and maintain the invariant $F_2 \Rrightarrow M_2$. Let us simplify $\overline{M}_2$, $\neg(x \geq 0 \wedge y \geq 0) \wedge \neg(y \leq 1)$, to $x < 0 \wedge y > 1$.

8. At this point $\overline{M}_2$ implies $x < 0$. So the next extrapolant C_3 will also imply $x < 0$. However, $\overline{M}_3$ cannot be satisfiable with C_3. We are done with level 3 and return *false* to level 2. In response, level 2 propagates the extrapolant $C_2 := (x < 0 \wedge y > 1)$ up to level 1.

9. Similar to step 4, $\overline{F}_1 \equiv \forall y.\neg\overline{F}_2$ implies that $\forall y.\neg C_2 \equiv \neg\exists y.C_2$. So if we take $\pi y.(C_2|\overline{M}_1)$ to be $x < 0$, then we can update $M_1 := \neg(x < 0)$, which is $x \geq 0$.

10. At this point let us check the levels below using $x \geq 0$. Then $\overline{M}_2 \wedge x \geq 0$ is unsatisfiable, so there is no refinement under this assumption. Return *false* to level 1. Level 1 is done, and we conclude the formula is satisfiable and an output quantifier-free formula is $x \geq 0$.

3 Extrapolation

A trivial extrapolant has been described in Section 2.1. We will here discuss two other versions of computing extrapolants $\langle A, B\rangle$.

3.1 SMT-TEST

The approach used in [9, 10] is to enumerate conjunctions of literals that satisfy $A \wedge B$. Suppose $\mathcal{L} := \ell_1, \ldots, \ell_n$ are the literals in the satisfying assignment for $A \wedge B$. An extrapolant is the intersection of $\mathcal{L}$ and an unsatisfiable core of $\mathcal{L} \wedge \neg B$. Our implementation of SMT-TEST extrapolation is using a single satisfiability check to extract a (not necessarily minimal) unsatisfiable core.

3.2 NNF strengthening

Algorithm 3: NNF extrapolant

```
$C \leftarrow NNF(B)$;
foreach literal $\ell \in C$ do
    $C' \leftarrow C[l/\ false]$;
    if $A \wedge C'$ is sat then
        $C \leftarrow C'$
    end
end
return $C$
```

NNF strengthening is a process of deriving a stronger formula by replacing literals by *false*. We start with a formula C which is a transformation of B to NNF. For each literal in C in order, replace that literal by *false* and check the conditions for extrapolation so that $\neg B \wedge C$ is `unsat` and $A \wedge C$ is `sat`. The first check is redundant (which holds by construction) and the second is not redundant. An extrapolant $C = \langle A,\ B\rangle$ is computed according to Algorithm 3. NNF strengthening gives us stronger formulas which potentially help reduce the number of iterations in procedure QT. We currently check satisfiability in each round during NNF strengthening. Another approach is to evaluate $C[l/\ false]$ using a model for C.

4 Projection specialized to Linear Integer Arithmetic

We are here interested in LIA since LIA decision procedure is used as the core of DCMC and plays a central role in feasibility of the model-checking approach [8]. Duration Calculus (DC) is an extension of Interval Temporal Logic with the notion of accumulated durations allowing succinct formulation of real-time problems. Chop ($\frown$) is the only modality appearing in DC; however, the model-checking problem in DC is transformed to satisfiability-checking of a LIA formula in size exponential to the chop-depth [7].

Cooper's algorithm for Presburger Arithmetic corresponds to quantifier elimination using *virtual substitutions*, and besides SMT-TEST for extrapolation we also consider strengthening formulas in negation normal form (NNF) by replacing literals by *false*. Virtual substitution methods work directly on formulas in NNF, so SMT-TEST is potentially not required.

4.1 Virtual substitutions

Virtual substitutions on LIA are performed by means of Cooper's algorithm. The algorithm removes quantifiers in the inside-out order using the following transformation:

$$\exists \mathtt{x}.\ \phi \iff \phi[\top/\mathtt{ax} < \mathtt{t},\ \bot/\mathtt{ax} > \mathtt{t}] \ \vee \ \bigvee_{\mathtt{i}=1}^{\delta} \bigvee_{\mathtt{ax}<\mathtt{t}} \phi[\mathtt{t}+\mathtt{i}/\mathtt{ax}] \wedge \delta' \mid \mathtt{t}+\mathtt{i}$$

where δ is the least common multiple of all divisors d in divisibility constraints $d \mid t$ and δ' is the least common multiple of all coefficients a in comparison constraints $ax \ \bowtie \ t$ where $a > 0$ and $\bowtie \in \{<, \leq, =, \geq, >\}$.

A quantified formula is transformed to a disjunction by a series of substitution steps. The disjunction can be represented symbolically (it corresponds to an existential quantifier over a finite domain) and may contain redundant disjuncts. The new divisibility constraint $\delta' \mid t + i$ could be replaced by many divisibility constraints of small divisors a in each substitution. In this setting, smaller divisors of inner formulas lead to fewer number of case splits for the next quantifier alternation.

Algorithm 4: $CS(\varphi)$

```
if $\varphi$ is unsat then
    return false
end
let $\mathcal{M}$ be a model for $\varphi$;
let $p$ be a fresh propositional variable;
assert $p \neq \varphi$;
return $CS(\varphi, \mathcal{M}, p)$
```

Algorithm 5: Auxiliary algorithm $CS(\varphi, \mathcal{M}, p)$

```
if $(p = \mathcal{M}(\varphi))$ is unsat in current context then
    return $\mathcal{M}(\varphi)$
end
foreach immediate subformula $\psi_i$ in $\varphi[\psi_1, \dots, \psi_k]$ do
    push;
    let $p_i$ be a fresh propositional variable;
    assert $\varphi[\psi_1, \dots, \psi_{i-1}, p_i, \psi_{i+1}, \dots, \psi_k] = p$;
    $\psi_i \leftarrow CS(\psi_i, \mathcal{M}, p_i)$;
    pop;
end
return $\varphi[\psi_1, \dots, \psi_k]$
```

4.2 Contextual simplification

Our projection procedure $\pi x.(C|M)$ admits using a context M when processing C. The simplification depends on the strength of M and it helps to trim down unnecessary cases in a virtual substitution method later. This process is called *contextual simplification* which is easy to implement in an SMT solver. Algorithm 4 contains a procedure for contextual simplification of formula φ. It works with a logical context, asserts that φ is unequal to p and recurses over sub-formulas of φ to replace them by *true* or *false*.

The approach also applies to non-Boolean domains (although this generalization is not required for QT): the auxiliary algorithm then takes terms of arbitrary types and checks if the terms are forced equal to the value provided in the model $\mathcal{M}$. Instead of recursing on sub-formulas it can recurse on sub-terms of arbitrary types. When φ is represented as a DAG and has exponentially many sub-formulas, the result can in fact get much larger than the input. A practical implementation of CS should therefore impose limits on how many times a sub-formula is traversed.

Contextual simplification is used in STeP [3] for simplifying formulas produced from verification condition generators and [5] develops a routine that works on formula trees in NNF, where it is used to speed up abstract interpreters that propagate formulas. Algorithm 4 is used in Z3 with the observation that a model $\mathcal{M}$ for φ can be used to prune checks for value forcing and it applies to subterms of arbitrary type. Z3 also contains cheaper contextual simplification routines that rely on accumulating equalities during a depth-first traversal. While this cheaper algorithm can replace sub-terms by constants, it is not necessarily an advantage to use in context of SMT solving: a non-constant sub-term can be much more useful for learning general purpose lemmas during search.

5 Evaluation

We implemented different combinations for the instantiated algorithm in the preview version of Z3 4.0. Projection procedures have been used including (A) - *full quantifier elimination* and (B) - *partial quantifier elimination.* There are three variants of extrapolation: (0) - *trivial extrapolation*, (1) - *NNF strengthening* and (2) *SMT-TEST.* Furthermore, we also implemented (X) - *contextual simplification* and (Y) - *no contextual simplification.* These components in order constitute 12 different combinations which are named in short as *aix* where $a \in \{A,\ B\}$, $i \in$

$\{0,\ 1,\ 2\}$ and $x \in \{X,\ Y\}$. For example, $A0Y$ denotes a combination of *trivial extrapolation*, *full quantifier elimination* and *no contextual simplification*.

We attempt to compare our algorithm (from now on called *AQS algorithm*) with Z3's quantifier elimination algorithm. Non-random benchmarks are collected from DCMC. Not only is the huge size (exponential to the chop-depth of DC formulas) of LIA formulas problematic, the nested nature between universal and existential quantifiers makes the problems even harder [8].

We divided benchmarks into two sets. **Set 1** consists of 32 *easy* formulas having from 56 to 94 quantifiers with file sizes ranging from 15KB to 33KB in SMT-LIB format. Z3's quantifier elimination can process these formulas within a few seconds. They are chosen for the purpose of recognizing incompetent candidates in 12 combinations above. **Set 2** have 64 *hard* instances with 69-768 quantifiers and take 50-500KB in SMT-LIB format. They are beyond the scope of Z3's quantifier elimination algorithm. We use them to test scalability of different combinations. Benchmarks and experimental results are available on the Z3 website [1]. The benchmark sets have some specific characteristics: coefficients are quite small and constraints are sparse (consisting of a few variables). These features help limit the number of disjunctions in virtual substitutions.

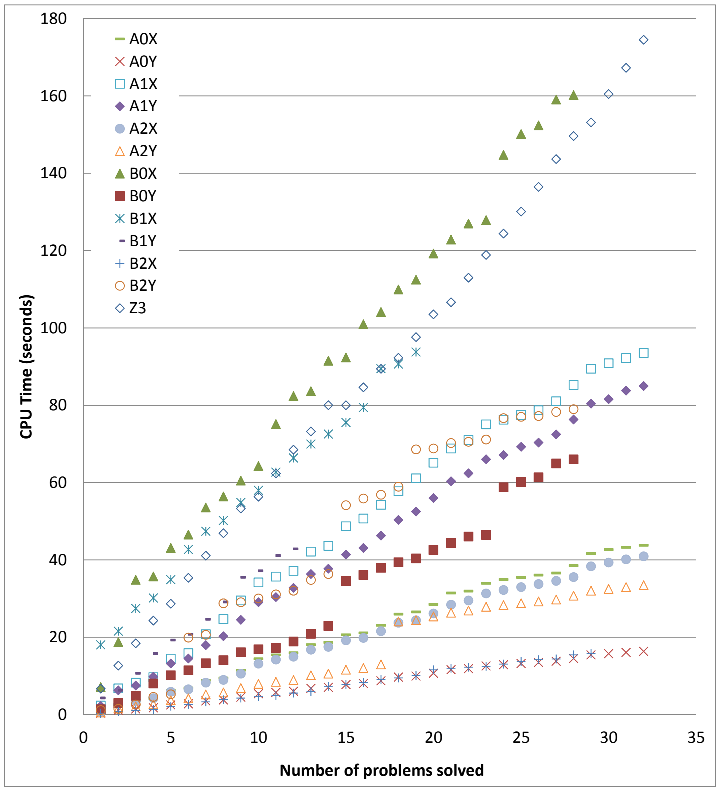

Figure 1: Accumulated running time of AQS vs. Z3 on benchmark set 1

[1] `http://research.microsoft.com/projects/z3/qt2012.zip`

Figure 1 summarizes the running time of different configurations of AQS as well as Z3's quantifier elimination algorithm. Each benchmark is given a 30-second timeout. The graph shows the accumulated running-time for solving all 32 problems. The winner is the configuration *A0Y* which means running AQS with projection implemented as *full quantifier elimination* and using *trivial extrapolation*. This configuration solves all benchmarks within 20 seconds. The configuration *A2Y* (using *SMT-TEST* instead of *trivial extrapolation*) is a close runner-up. It also solves all benchmarks, but requires 35 seconds. This benchmark set is simply too small to draw clear conclusions between these approaches. Partial quantifier elimination (configurations with prefix *B*) is bad on all configurations. The experiments also suggest that strong context simplification is pure overhead in all configurations. In an earlier prototype outside of Z3, however, strong context simplification was an advantage. We attribute this to how constraints get simplified and subsumed when being passed between AQS and Z3's quantifier elimination procedure. Z3's built-in quantifier elimination procedure has a slower ramp up time and is able to solve all problems within 175 seconds.

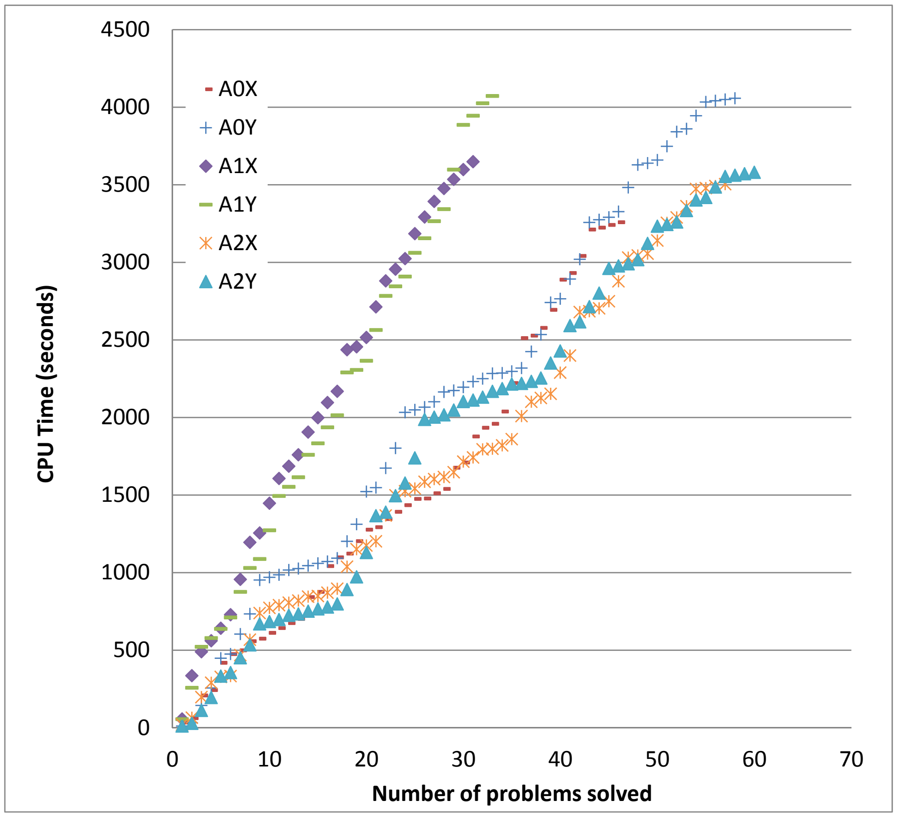

Figure 2: Accumulated running time of AQS on benchmark set 2

Experiment 2 was performed on **Set 2** for all configurations of AQS with timeout of 300 seconds. Experimental results are shown in Figure 2. We omit running time for configurations with partial quantifier elimination, they solve almost no problems, and we also omit running time for the default quantifier elimination routine that also does not solve any problem. The experimental results indicate that AQS algorithm performs well on formulas with many blocks of nested quantifiers. We have gained an order of magnitude speedup for Duration Calculus application. The *A2Y* (using *full projection* and *SMT-TEST*) configuration scales well on our benchmarks although *A0Y* (using *full projection* and *trivial extrapolation*) comes intriguingly close.

6 Conclusions

We presented an anatomy of the algorithm proposed in [10] for checking satisfiability of formulas with alternating quantification. We proposed a set of generalizations, applied them to Presburger Arithmetic, and evaluated the generalizations to benchmarks from a model checker for Duration Calculus. So far the experience has been that the satisfiability algorithms, when instantiated with SMT-TEST (and to some extent trivial extrapolation) perform orders of magnitude better than general purpose quantifier elimination. We are currently investigating additional alternatives to the algorithms presented here. One alternative is to instantiate quantifiers incrementally using virtual substitutions. The idea is similar to how quantifiers are instantiated using E-matching in SMT solvers. SMT solvers create a propositional abstraction of formulas, including quantifiers. If there is a satisfying assignment to the abstracted formula that does not depend on the quantified sub-formulas, then the formula is satisfiable. Otherwise, quantified formulas are *model-checked* with respect to the current model and only instantiated (by axioms of the form "$(\forall x \varphi[x]) \Rightarrow \varphi[t]$") when the interpretation for free variables cannot be extended to an interpretation that satisfies the quantified formulas.

Acknowledgments

This work grew out of a course on SMT solving organized by Flemming Nielson and Hanne R. Nielson. We are grateful for valuable comments and permission of using DCMC benchmarks from Michael R. Hansen. Anh-Dung Phan is supported by the IDEA4CPS project granted by the Danish Research Foundation for Basic Research.

References

[1] Armin Biere. Resolve and Expand. In Holger H. Hoos and David G. Mitchell, editors, *SAT (Selected Papers*, volume 3542 of *Lecture Notes in Computer Science*, pages 59–70. Springer, 2004.

[2] Nikolaj Bjørner. Linear Quantifier Elimination as an Abstract Decision Procedure. In Jürgen Giesl and Reiner Hähnle, editors, *IJCAR*, volume 6173 of *Lecture Notes in Computer Science*, pages 316–330. Springer, 2010.

[3] Nikolaj Bjørner, Anca Browne, Edward Y. Chang, Michael Colón, Arjun Kapur, Zohar Manna, Henny Sipma, and Tomás E. Uribe. STeP: Deductive-Algorithmic Verification of Reactive and Real-Time Systems. In Rajeev Alur and Thomas A. Henzinger, editors, *CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 415–418. Springer, 1996.

[4] D. Cooper. Theorem proving in arithmetic without multiplication. In *Machine Intelligence*, 1972.

[5] Isil Dillig, Thomas Dillig, and Alex Aiken. Small formulas for large programs: On-line constraint simplification in scalable static analysis. In Radhia Cousot and Matthieu Martel, editors, *SAS*, volume 6337 of *Lecture Notes in Computer Science*, pages 236–252. Springer, 2010.

[6] Michael J. Fischer and Michael O. Rabin. Super-Exponential Complexity of Presburger Arithmetic. In *Proceedings of the SIAM-AMS Symposium in Applied Mathematics*, 1974.

[7] Martin Fränzle and Michael R. Hansen. Efficient Model Checking for Duration Calculus? *Int. J. Software and Informatics*, 3(2-3):171–196, 2009.

[8] Michael R. Hansen and Aske Wiid Brekling. On Tool Support for Duration Calculus on the basis of Presburger Arithmetic. In Carlo Combi, Martin Leucker, and Frank Wolter, editors, *TIME*, pages 115–122. IEEE, 2011.

[9] David Monniaux. A Quantifier Elimination Algorithm for Linear Real Arithmetic. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *LPAR*, volume 5330 of *Lecture Notes in Computer Science*, pages 243–257. Springer, 2008.

[10] David Monniaux. Quantifier Elimination by Lazy Model Enumeration. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 585–599. Springer, 2010.

[11] Derek C. Oppen. A $2^{2^{2^{pn}}}$ Upper Bound on the Complexity of Presburger Arithmetic. *J. Comput. Syst. Sci.*, 16(3):323–332, 1978.

[12] David A. Plaisted, Armin Biere, and Yunshan Zhu. A satisfiability procedure for quantified Boolean formulae. *Discrete Applied Mathematics*, 130(2):291–328, 2003.

[13] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, pages 4–13, New York, NY, USA, 1991. ACM.

[14] Ryan Stansifer. Presburgerś Article on Integer Airthmetic: Remarks and Translation. Technical report, Cornell University, Computer Science Department, September 1984.