E-matching for Fun and Profit

Michał Moskal[1,2] Jakub Łopuszański[3]

Institute of Computer Science
University of Wrocław
Wrocław, Poland

Joseph R. Kiniry[4]

University College Dublin
Dublin, Ireland

Abstract

Efficient handling of quantifiers is crucial for solving software verification problems. E-matching algorithms are used in satisfiability modulo theories solvers that handle quantified formulas through instantiation. Two novel, efficient algorithms for solving the E-matching problem are presented and compared to a well-known algorithm described in the literature.

Keywords: E-matching, quantifier instantiation, SMT.

1 Motivation

Satisfiability Modulo Theories (SMT) solvers usually operate in the quantifier-free fragments of their respective logics. Yet program verification problems often require expressiveness and flexibility in extending the underlying background theories with universally quantified axioms. The typical solution to this problem is to generate ground instances of the quantified subformulas during the course of the proof search and hope that the particular instances generated are the ones required to prove unsatisfiability.

As an example, consider the following formula, which we try to satisfy modulo linear arithmetic and uninterpreted function symbols theories:

$$P(f(42)) \wedge \forall x.\, P(f(x)) \Rightarrow x < 0$$

[1] Partially supported by Polish Ministry of Science and Education grant 3 T11C 042 30.

[2] Email: mjm@ii.uni.wroc.pl

[3] Email: jlo@ii.uni.wroc.pl

[4] Email: kiniry@acm.org

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

If the prover were able to guess the implication:

$$(\forall x.\, P(f(x)) \Rightarrow x < 0) \Rightarrow P(f(42)) \Rightarrow 42 < 0$$

then, by boolean unit resolution (with the currently known facts $P(f(42))$ and $\forall x.\, P(f(x)) \Rightarrow x < 0$), the prover would try to assert $42 < 0$, which would cause contradiction in the linear arithmetic decision procedure.

The tricky part is how to figure out which instances are going to be useful. A well-known [7] solution is to designate subterms occurring in the quantified formula called *triggers*, and only add instances that make those subterms equal to ground subterms that are currently being considered in the proof. In our example one such trigger is $P(f(x))$, which works as expected.

However it is often not enough to consider only syntactic equality. If we modify our example a little bit:

$$a = f(42) \wedge P(a) \wedge \forall x.\, P(f(x)) \Rightarrow x < 0$$

then our choice of trigger no longer works. We could use a less restrictive trigger (namely $f(x)$), but such a trigger leads to generating excessive, irrelevant instances, which reduces the efficiency of the prover. We therefore use a different technique: instead of using syntactic equality, use the equality relation induced by the current context [5]. For example: in the context $P(a)$, $a = f(42)$ the substitution $[x := 42]$ makes the term $P(f(x))$ equal to $P(a)$.

Because we do not treat boolean formulas as terms, it is sometimes not possible to designate a single trigger containing all the variables that are quantified. A classical example is the transitivity axiom. In such a case we use a *multitrigger*, which is a set of triggers, hopefully sharing variables, that are supposed to match simultaneously.

There are two remaining problems here: identifying the set of triggers for a given formula, and identifying the substitutions that make the trigger equal to some ground term. As for the first problem, it is possible to apply heuristics [6], as well as ask the user to provide the triggers. The second problem is E-matching. We present a well-known algorithm for solving the E-matching problem (Sect. 3), introduce two other, efficient algorithms (Sect. 4 and 5) and compare them to the well-known one.

2 Definitions

Let $\mathcal{V}$ be the infinite, enumerable set of variables. Let Σ be the set of function and constant symbols. Let $\mathcal{T}$ be the set of first order terms constructed over Σ and $\mathcal{V}$.

A *substitution* is a function $\sigma : \mathcal{V} \to \mathcal{T}$ that is not the identity only for a finite subset of $\mathcal{V}$. We identify a substitution with its homomorphic extension to all terms (i.e., $\sigma : \mathcal{T} \to \mathcal{T}$). Let $\mathcal{S}$ be the set of all substitutions.

We will use letters x and y, possibly with indices for variables, f and g for function symbols, c and d for constant symbols (functions of arity zero), σ and ρ for substitutions, t for ground terms, and p for possibly non-ground terms. We

[5] SMT solvers usually refute a formula ψ by refuting every boolean assignment to literals of ψ that make ψ true. The term "current context" refers to such an assignment (which can be partial).

[6] Some heuristics are described in the Simplify technical report [7].

will use the notation $[x_1 := t_1, \ldots, x_n := t_n]$ for substitutions, and $\sigma[x := t]$ for a substitution augmented to return t for x.

An instance of an *E-matching problem* [7] consists of a finite set of *active* ground terms $\mathcal{A} \subseteq \mathcal{T}$, a relation $\cong_g \subseteq \mathcal{A} \times \mathcal{A}$, and a finite set of non-variable, non-constant terms $p_1, \ldots, p_n$, which we call the *triggers*. Let $\cong \subseteq \mathcal{T} \times \mathcal{T}$ be the smallest congruence relation over Σ containing $\cong_g$. Let $\texttt{root} : \mathcal{T} \to \mathcal{T}$ be a function [8] such that:

$$(\forall t, s \in \mathcal{T}.\, \texttt{root}(t) = \texttt{root}(s) \Leftrightarrow t \cong s) \wedge (\forall t \in \mathcal{T}.\, \texttt{root}(t) \cong t)$$

The solution to the E-matching problem is the set:

$$T = \left\{ \sigma \;\middle|\; \begin{array}{l} \exists t_1, \ldots, t_n \in \mathcal{A}.\, \sigma(p_1) \cong t_1 \wedge \ldots \wedge \sigma(p_n) \cong t_n, \\ \forall x \in \mathcal{V}.\, \sigma(x) = \texttt{root}(\sigma(x)) \end{array} \right\}$$

The problem of deciding for a fixed $\mathcal{A}$ and $\cong_g$, and a given trigger, if $T \neq \emptyset$, is NP-hard [12]. The NP-hardness is why each solution to the problem is inherently backtracking in nature. In practice, though, the triggers that are used are small, and the problem is not the complexity of a backtracking search for a particular trigger, but rather the fact that in a given proof search there are often hundreds of thousands of matching problems to solve.

3 Simplify's Matching Algorithm

Simplify is a legacy SMT system, the first one to efficiently combine theory and quantifier reasoning. This combination made it a popular target for various software verification systems. The Simplify technical report [7] describes a recursive matching algorithm *simplify_match* given in Fig. 1. The symbol :: denotes a list constructor, *nil* is an empty list, $[x_1, \ldots, x_n]$ is a shorthand for $x_1 :: \ldots :: x_n :: nil$, and [] is an empty (identity) substitution. *hd* and *tl* are the functions returning, respectively, head and tail of a list (i.e., $hd(x :: y) = x$ and $tl(x :: y) = y$). The command **skip** is a no-op.

The *simplify_match* algorithm maintains the current substitution and a stack (implemented as a list) of (trigger, ground term) pairs to be matched. We refer to these pairs as *jobs*. Additionally, it uses the special variable $*$ in place of a ground term to say that we are not interested in matching against any specific term, as any active term will do.

We start (line marked /* 4 */) by putting the set of triggers to be matched on the stack and then proceed by taking the top element of the stack.

If the trigger in the top element is a constant (/* 1 */), we just compare it against the ground term, and if the comparison succeeded, recurse.

If the trigger is a variable x (/* 2 */), we check if the current substitution already assigns some value to that variable, and if so, we just compare it against the ground term t. Otherwise, we extend the current substitution by mapping x to t and

[7] In the automated reasoning literature, the term E-matching usually refers to a slightly different problem, where $\mathcal{A}$ is a singleton and $\cong$ is not restricted to be finitely generated [15]. On the other hand the Simplify technical report [7] as well as the recent Z3 paper [6] use the term E-matching in the sense defined above.

[8] Such a function exists by virtue of $\cong$ being an equivalence relation, and is provided by the typical data structure used to represent $\cong$, namely the E-graph (see Simplify technical report [7] for details on E-graph).

```
fun simplify_match([p_1, ..., p_n])
  R := ∅
  proc match(σ, j)
    if j = nil then R := R ∪ {σ}
    else case hd(j) of
      (c, t) ⇒      /* 1 */
        if c ≅ t then match(σ, tl(j))
        else skip
      (x, t) ⇒      /* 2 */
        if σ(x) = x then match(σ[x := root(t)], tl(j))
        else if σ(x) = root(t) then match(σ, tl(j))
        else skip
      (f(p_1, ..., p_n), t) ⇒      /* 3 */
        foreach f(t_1, ..., t_n) in A do
          if t = * ∨ root(f(t_1, ..., t_n)) = t then
            match(σ, (p_1, root(t_1)) :: ... :: (p_n, root(t_n)) :: tl(j))
  match([], [(p_1, *), ..., (p_n, *)])      /* 4 */
  return R
```

Fig. 1. Simplify's matching algorithm

recurse. Observe that t cannot be $*$ because we do not allow a trigger to be a single variable, and $*$ is only paired with triggers in the initial call, never with subtriggers.

If the trigger is a complex term $f(p_1, \ldots, p_n)$ (/* 3 */), we iterate over all the terms with f in the head (possibly checking if they are equivalent to the ground term we are supposed to match against), construct the set of jobs matching respective children of the trigger against respective children of the ground term, and recurse.

The important invariants of *simplify_match* are: (1) the jobs lists contain stars instead of ground terms only for non-variable, non-constant triggers; (2) all the ground terms t in the job lists satisfy $\texttt{root}(t) = t$; (3) for all x either $\sigma(x) = x$ or $\sigma(x) = \texttt{root}(\sigma(x))$.

The detailed discussion of this procedure is given in the Simplify technical report [7].

4 Subtrigger Matcher

This section describes a novel matching algorithm, optimized for linear triggers. A *linear trigger* is a trigger in which each variable occurs at most once. Most triggers used in the program verification problems we have inspected are linear. The linearity means that matching problems for subterms of a trigger are independent, which allows for more efficient processing.

However, even if triggers are not linear, it pays off to treat them as linear, and only after the matching algorithm is complete discard the resulting substitutions that assign different terms to the same variable. This technique is often used in

fun $fetch(S, t, p)$
  if $S = \top$ **then return** $\{[p := \texttt{root}(t)]\}$
  else if $S = \times \wedge t \cong p$ **then return** $\{[]\}$
  else if $S = \times$ **then return** $\emptyset$
  else return $S(\texttt{root}(t))$

fun $match(p)$
  case p **of**
    $x \Rightarrow$ **return** $\top$
    $c \Rightarrow$ **return** $\times$
    $f(p_1, \ldots, p_n) \Rightarrow$
      foreach i **in** $1 \ldots n$ **do** $S_i := match(p_i)$ /* 1 */
      if $\exists i.\, S_i = \bot$ **then return** $\bot$ /* 2 */
      if $\forall i.\, S_i = \times$ **then return** $\times$ /* 3 */
      $S := \{t \mapsto \emptyset \mid t \in \mathcal{A}\}$
      foreach $f(t_1, \ldots, t_n)$ **in** $\mathcal{A}$ **do** /* 4 */
        $t := \texttt{root}(f(t_1, \ldots, t_n))$
        $S := S[t \mapsto S(t) \sqcup (fetch(S_1, t_1, p_1) \sqcap \ldots \sqcap fetch(S_n, t_n, p_n))]$
      if $\forall t.\, S(t) = \bot$ **then return** $\bot$
      else return S

fun $topmatch(p)$ /* 5 */
  $S := match(p)$
  return $\bigsqcup_{t \in \mathcal{A}} S(t)$

fun $subtrigger\_match([p_1, \ldots, p_n])$
  return $topmatch(p_1) \sqcap \ldots \sqcap topmatch(p_n)$

Fig. 2. Subtrigger matching algorithm

term indexes [16] used in automated reasoning. The algorithm, therefore, does not require the trigger to be linear.

This matcher algorithm is given in Fig. 2. It uses operations $\sqcap$ and $\sqcup$, which are defined on sets of substitutions:

$$A \sqcap B = \{\sigma \oplus \rho \mid \sigma \in A, \rho \in B, \sigma \oplus \rho \neq \bot\}$$

$$A \sqcup B = A \cup B$$

$$\sigma \oplus \rho = \begin{cases} \bot & \text{when } \exists x.\, \sigma(x) \neq x \wedge \rho(x) \neq x \wedge \sigma(x) \neq \rho(x) \\ \sigma \cdot \rho & \text{otherwise} \end{cases}$$

$$\sigma \cdot \rho(x) = \begin{cases} \sigma(x) & \text{when } \sigma(x) \neq x \\ \rho(x) & \text{otherwise} \end{cases}$$

$\sqcap$ returns a set of all possible non-conflicting combinations of two sets of substitutions. $\sqcup$ sums two such sets. The next section shows an implementation of these operations that does not use explicit sets.

The *match(p)* function returns the set of all substitutions σ, such that $\sigma(p) \cong t$, for a term $t \in \mathcal{A}$, categorized by $\texttt{root}(t)$. More specifically, *match* returns a map from $\texttt{root}(t)$ to such substitutions, or one of the special symbols $\top$, $\bot$, $\times$. Symbol $\top$ means that p was a variable x, and therefore the map is: $\{t \mapsto \{[x := t]\} \mid t \in \mathcal{A},\ \texttt{root}(t) = t\}$, symbol $\bot$ represents no matches (i.e., $\{t \mapsto \emptyset \mid t \in \mathcal{A}, \texttt{root}(t) = t\}$), and $\times$ means p was ground, so the map is $\{\texttt{root}(p) \mapsto \{[]\}\} \cup \{t \mapsto \emptyset \mid t \in \mathcal{A}, t = \texttt{root}(t), t \neq p\}$ [9].

The only non-trivial control flow case in the *match* function is the case of a complex trigger $f(p_1, \ldots, p_n)$, which works as follows:

- /* 1 */ recurse on subtriggers. Conceptually, we consider the subtriggers to be independent of each other (i.e., $f(p_1, \ldots, p_n)$ is linear). If they are, however, dependent, then the $\sqcap$ operation filters out conflicting substitutions.
- /* 2 */ check if there is any subtrigger that does not match anything, in which case the entire trigger does not match anything.
- /* 3 */ check if all the children of $f(p_1, \ldots, p_n)$ are ground, in which case $f(p_1, \ldots, p_n)$ is ground as well.
- /* 4 */ otherwise we start with an empty result map S and iterate over all terms with the correct head symbol. For each such term $f(t_1, \ldots, t_n)$, we combine (using $\sqcup$) the already present results for $\texttt{root}(f(t_1, \ldots, t_n))$ with results of matching p_i against t_i. The *fetch* function is used to retrieve results of subtrigger matching by ensuring the special symbols are treated as the maps they represent.

Finally (/* 5 */) the *topmatch* function just collapses the maps into one big set.

4.1 S-Trees

This section introduces a new data structure, s-tree, which is used to compactly represent sets of substitutions, so they can be efficiently manipulated during the matching.

The s-trees data structure itself can be viewed as a special case of substitution trees used in automated reasoning [16] with rather severe restrictions on their shape. We, however, do not use the trees as an index and, as a consequence, require a different set of operations on s-trees than those defined on substitution trees.

S-trees require a strict, total order $\prec \subseteq \mathcal{A} \times \mathcal{A}$ and are defined inductively: (1) ϵ is an s-tree; (2) if $T_1, \ldots, T_n$ are s-trees and $t_1, \ldots, t_n$ are ground terms, then the pair $x \rhd ([(t_1, T_1), \ldots, (t_n, T_n)])$ is an s-tree.

The invariant of the s-tree data structure is that in each node the term $t_1, \ldots, t_n$ are sorted according to $\prec$ (i.e. for all i and j, $i < j \Rightarrow t_i \prec t_j$) [10], and that there exists a sequence of variables $x_1 \ldots x_k$ such that the root has the form $x_1 \rhd (\ldots)$

[9] Here we assume all the ground subterms of triggers to be in $\mathcal{A}$. This is easily achieved and does not affect performance in our tests.

[10] This invariant is employed in implementation of the $\sqcup$ operator.

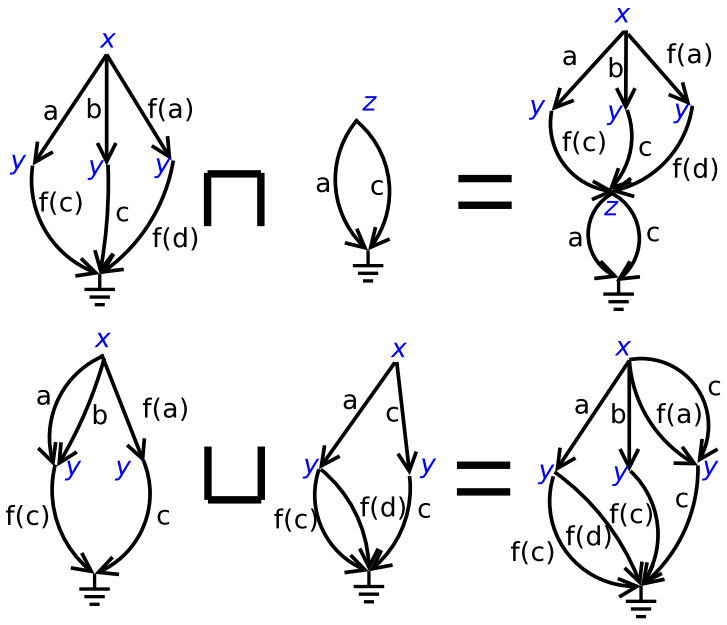

Fig. 3. Example of s-tree operations

and each node (including the root) has the form:

(i) $x_i \rhd ([(t_1, x_{i+1} \rhd (\ldots)), \ldots, (t_n, x_{i+1} \rhd (\ldots))])$ or

(ii) $x_k \rhd ([(t_1, \epsilon), \ldots, (t_n, \epsilon)])$

for some n, $t_1, \ldots, t_n$ and $1 \leq i < k$. In other words, the variables at a given level of a tree are the same.

The *yield* function maps a s-tree into the set of substitutions it is intended to represent.

$$yield(\epsilon) = \{[]\}$$

$$yield(x \rhd ([(t_1, T_1), \ldots, (t_n, T_n)])) = \left\{ \sigma[x := t_i] \;\middle|\; \begin{array}{l} i \in \{1, \ldots, n\}, \\ \sigma \in yield(T_i) \\ \sigma(x) = x \vee \sigma(x) = t_i \end{array} \right\}$$

Example s-trees are given in Fig. 3. The trees are represented as ordered directed acyclic graphs with aggressive sharing. An s-tree $x \rhd ([(t_1, T_1), \ldots, (t_n, T_n)])$ has the label x on the node, t_i label the edges and each edge leads to another tree T_i. The ground symbol corresponds to the empty tree ϵ. E.g., the middle bottom one represents $x \rhd ([(a, y \rhd ([(f(c), \epsilon), (f(d), \epsilon)])), (c, y \rhd ([(c, \epsilon)]))])$, which yields $\{[x := a, y := f(c)], [x := a, y := f(d)], [x := c, y := c]\}$. The ordering of terms used in the example is $a \prec b \prec c \prec d \prec f(a) \prec f(c) \prec f(d)$.

We now define an analogous for s-trees of the operators $\sqcup$ and $\sqcap$ we defined earlier for sets of substitutions. Formally, the operators are defined so that *yield* is

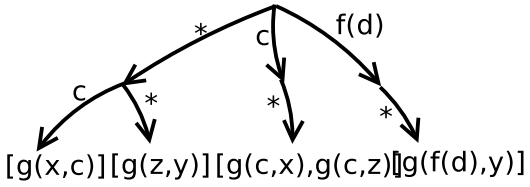

Fig. 4. Example of an index for flat triggers with g in head

a homomorphism from s-trees to sets of substitutions.

$$\epsilon \sqcap T = T$$

$$x \rhd ([(t_1, T_1), \ldots, (t_n, T_n)]) \sqcap T = x \rhd ([(t_1, T_1 \sqcap T), \ldots, (t_n, T_n \sqcap T)])$$

$$\epsilon \sqcup \epsilon = \epsilon$$

$$x \rhd (X) \sqcup x \rhd (Y) = x \rhd (merge(X, Y))$$

$$merge((t, T) :: X, (t', T') :: Y) = \begin{cases} (t, T \sqcup T') :: merge(X, Y) & \text{if } t = t' \\ (t, T) :: merge(X, (t', T') :: Y) & \text{if } t < t' \\ (t', T') :: merge((t, T) :: X, Y) & \text{if } t' < t \end{cases}$$

$$merge(nil, X) = X$$

$$merge(X, nil) = X$$

The $\sqcap$ corresponds to stacking trees one on top of another, while $\sqcup$ does a recursive merge. Example applications are given in Fig. 3.

The precondition of the $\sqcup$ operator is that the operands have the same shape, meaning the $x_1 \ldots x_k$ sequence from the invariant is the same for both trees; otherwise, $\sqcup$ is undefined. This precondition is fulfilled by the subtrigger matcher, since it only combines trees resulting from matching of the same trigger, which means the variables are always accessed in the same order.

To change *subtrigger_match* to use s-trees, we need to change the *fetch* function, to return $p \rhd ([(\texttt{root}(t), \epsilon)])$ instead of $[p := \texttt{root}(t)]$, ϵ instead of $\{[]\}$ and $x \rhd (nil)$ for some variable x instead of $\emptyset$. After this is done we only call *yield* at the very end, to transform an s-tree into a set of substitutions.

5 Flat Matcher

During performance testing, we found that most triggers shared the head symbol and matching them was taking a considerable amount of time. Moreover, the triggers had a very simple form: $f(x, c)$ [11] . This form is a specific example of something we call flat triggers. A *flat trigger* is a trigger in which each variable occurs at most once and at a depth of one.

Flat triggers with a given head can be matched all at once by constructing a tree that indexes all the triggers with the given function symbol in the head. Such a tree can be viewed as a special kind of a discrimination tree [16], where we consider each

[11] The actual function symbol was a subtyping predicate in ESC/Java2's [11] Simplify-based object logic.

```
fun topmatch(p')
  if p is flat then
    let f(p_1, ..., p_n) = p
    I_f := index for all triggers with head f
    foreach p in I_f do S_p := ∅
    foreach f(t_1, ..., t_n) in A do
      foreach p ↦ T in match'(f(t_1, ..., t_n), [t_1, ..., t_n], I_f) do
        S_p := S_p ⊔ T
    return S_p'
  else
    S := match(p')
    return ⨆_{t∈A} S(t)
```

Fig. 5. Flat matcher

child of the pattern as a constant term, instead of traversing it pre-order. Unlike in discrimination trees used for matching our index has non-ground terms and queries are ground.

We assume, without loss of generality, each function symbol to have only one arity. A node in the index tree is either a set of triggers $\{p_1, \ldots, p_n\}$, or a set of pairs $\{(t_1, I_1), \ldots, (t_n, I_n)\}$, where each of the t_i is a ground term or a special variable $*$, and I_i are index trees.

We call $(t_1, \ldots, t_n, p)$ a *path* in I if and only if: (1) $n = 0$ and $p \in I$; or (2) $(t_1, I') \in I$ and $(t_2, \ldots, t_n, p)$ is a path in I'.

Let $star(x) = *$ for a variable x and $star(t) = t$, for any non-variable term t. We say that I indexes a set of triggers Q if for any $f(p_1, \ldots, p_n) \in Q$ there exists a path $(star(p_1), \ldots, star(p_n), f(p_1, \ldots, p_n))$ in I, and for every path there exists a corresponding trigger.

Given an index I, we find all the triggers that match the term $f(t_1, \ldots, t_n)$ by calling $match'(f(t_1, \ldots, t_n), [t_1, \ldots, t_n], \{I\})$, where $match'$ is defined as follows:

$$
\begin{aligned}
&match'(t, [t_1, \ldots, t_n], A) = \\
&\quad match'(t, [t_2, \ldots, t_n], \{I' \mid I \in A, (p, I') \in I,\ p \cong t_1 \vee p = *\}) \\
&match'(f(t_1, \ldots, t_n), nil, A) = \\
&\quad \{f(p_1, \ldots, p_n) \mapsto \sqcap_{i=1\ldots n,\, p_i \in \mathcal{V}} [p_i := \texttt{root}(t_i)] \ \mid\ I \in A, f(p_1, \ldots, p_n) \in I\}
\end{aligned}
$$

The algorithm works by maintaining the set of trigger indices A containing triggers that still possibly match t. At the bottom of the tree we extract the children of t corresponding to variables in the trigger. We only consider position at which the trigger has variables, not ground terms (the condition $p_i \in \mathcal{V}$).

A flat-aware matcher is implemented by replacing the *topmatch* function from Fig. 2 with the one from Fig. 5. The point of using it, though, is to cache I_f and S_p across calls to *subtrigger_match*.

6 Implementation and Experiments

We have implemented all three algorithms inside the Fx7 SMT solver [12]. Fx7 is implemented in the Nemerle [13] language and runs on the .NET platform. In each case the implementation is highly optimized and only unsatisfactory results with the *simplify_match* algorithm led to designing and implementing the second and the third algorithm.

The implementation makes heavy use of memoization. Both terms and s-trees use aggressive (maximal) sharing. The implementations of $\sqcap$ and $\sqcup$ exploit this sharing, by memoizing results to avoid processing the same (shared) subtree more than once.

An important point to consider in the design of matching algorithms is incrementality. The prover will typically match, assert some facts, and then match again. The prover is then interested only in receiving the new results. The Simplify technical report [7] cites two optimizations to deal with incrementality. We have implemented one of them, the *mod-time* optimization, in all three algorithms. The effects are mixed, mainly because our usage patterns of the matching algorithm are different than those of Simplify: we generally change the E-graph more between matchings due to our proof search strategy.

To achieve incrementality we memoize s-trees returned on a given proof path and then use the subtraction operation on s-trees to remove substitutions that had been returned previously. The subtraction on s-trees corresponds to set subtraction and its implementation is very similar to the one of $\sqcup$.

Another fine point is that the loop over all active terms in the implementations of all three algorithms skips some terms: if we have inspected $f(t_1, \ldots, t_n)$ then we skip $f(t'_1, \ldots, t'_n)$ given that $t_i \cong t'_i$ for $i = 1 \ldots n$. Following work on fast, proof-producing congruence closure [14], we encode all the terms using only constants and a single binary function symbol $\cdot(\ldots)$. E.g., $f(t_1, \ldots, t_n)$ is represented by $\cdot(f, \cdot(t_1, \ldots \cdot (t_{n-1}, t_n)))$. Therefore the loop over active terms is skipped when $\texttt{root}(\cdot(t_1, \ldots \cdot (t_{n-1}, t_n)))$ was already visited.

Yet another issue is that we map all the variables to one special symbol during the matching, do not store the variable names in s-trees, and only introduce the names when iterating the trees to get the final results (inside the *yield* function). This allows for more sharing of subtriggers between different triggers and is fairly common practice in term indexing.

The tests were performed on a 1 GHz Pentium III box with 512 MiB of RAM running Linux and Nemerle r7446 on top of Mono 1.2.3. The memory used was always under 200 MiB. We ran the prover on a randomly selected set of verification queries generated by the ESC/Java [10] and Boogie [2] tools. The benchmarks are now available as part of SMT-LIB [17].

The subtrigger matcher helps speed up matching by around 20% in the Boogie benchmarks and around 50% in the ESC/Java benchmarks. The flat matcher is around 2 times faster than Simplify's matcher in the Boogie benchmarks and around 10 times in the ESC/Java benchmarks. The detailed results are given in the

[12] Available online at http://nemerle.org/fx7/.

Appendix A.

Now we give some intuitions behind the results. For example, consider the trigger $f(g_1(x_1), \ldots, g_n(x_n))$. If each of $g_i(x_i)$ returns two matches, except for the last one which does not match anything, the subtrigger matcher exits after $O(n)$ steps, while the Simplify matcher performs $O(2^n)$ steps. Even when $g_n(x_n)$ actually matches something (which is more common), the subtrigger algorithm still performs $O(n)$ steps to construct the s-tree and only performs $O(2^n)$ steps walking that tree. These steps are much cheaper (as the tree is rather small and fits in the CPU cache) than matching the g_is several times, which Simplify's algorithm does. The main point of the subtrigger matcher is therefore not to repeat work for a given (sub)trigger more than once.

The benefits of the flat matcher seem to be mostly CPU cache-related. For example, a typical problem might have one hundred triggers with head f, and one thousand ground terms with the head f. The flat matcher processes a data structure (of size one hundred) one thousand times, while the subtrigger matcher (and also Simplify's matcher) processes a different data structure (of size one thousand) one hundred times. Consequently, given that these data structures occupy a considerable amount of memory, frequently the smaller data structures in the former case fit the cache, while the larger ones in the latter case do not.

7 Conclusions and Related Work

We have presented two novel algorithms for E-matching. They are shown to outperform the well-known Simplify E-matching algorithm.

The E-matching problem was first described, along with a solution, in the Simplify technical report [7]. We know several SMT solvers, like Zap [1], CVC3 [4], Verifun [9], Yices [8] and Ergo [5] include matching algorithms, though there seem to be no publications describing their algorithms. Specifically, Zap uses a different algorithm that also relies on the fact of triggers being linear and uses a different kind of s-trees. Zap, however, does not do anything special about flat triggers.

In a recent paper [6] on Z3 (a rewrite of the Zap prover), a way of compiling patterns into a code tree that is later executed against ground terms is defined. Such a tree is beneficial if there are many triggers that share the top part of triggers. We, on the other hand, exploit sharing in the bottom parts of triggers, and the flat matcher handles the case of simple triggers that share only the head symbol. The Z3 authors also propose an index on the ground terms that is used to speed up matching in an incremental usage pattern. Such an index could perhaps be used also with our approach. Of course, the usefulness of all these techniques largely depends on benchmarks and the particular search strategy employed in an SMT solver.

During the 2007 SMT competition [3] there were four solvers participating in the Arithmetic, Uninterpreted Functions, Linear Integer Arithmetic and Arrays (AUFLIA) division. The AUFLIA division includes software verification problems and is the only one involving heavy use of quantifier reasoning. Z3 was first and Fx7 was second, with the same number of solved benchmarks but much worse run time. Both solvers used improved matching algorithms, while other participants (CVC3

and Yices) did not, which is some indication of importance of E-matching in this kind of benchmarks.

Some of the problems in the field of term indexing [16] in saturation-based theorem provers are also related. As mentioned earlier, our work uses ideas similar to substitution trees and discrimination trees. It seems to be the case, however, that the usage patterns in the saturation provers are different than those in SMT solvers. Matching SMT solvers must deal with several orders of magnitude fewer non-ground terms, a similar number of ground terms, but the time constraints are often much tighter. This different set of constraints and goals consequently leads to the construction of different algorithms and data structures.

We would like to thank Mikoláš Janota for his comments regarding this paper.

This work is being supported by the European Project Mobius within the frame of IST 6th Framework, national grants from the Science Foundation Ireland and Enterprise Ireland, and by the Irish Research Council for Science, Engineering and Technology. This paper reflects only the authors' views and the Community is not liable for any use that may be made of the information contained therein. This work is partially supported by Science Foundation Ireland under grant number 03/CE2/I303-1, "LERO: the Irish Software Engineering Research Centre."

References

[1] Ball, T., S. K. Lahiri and M. Musuvathi, *Zap: Automated theorem proving for software analysis.*, in: G. Sutcliffe and A. Voronkov, editors, *LPAR*, Lecture Notes in Computer Science **3835** (2005), pp. 2–22.

[2] Barnett, M., K. Leino and W. Schulte, *The Spec# programming system: An overview*, in: *Proceeding of CASSIS 2004*, LNCS **3362** (2004).

[3] Barrett, C., L. de Moura and A. Stump, *Design and Results of the 3rd Satisfiability Modulo Theories Competition (SMT-COMP 2007)* (2007), to appear, also http://www.smtcomp.org/2007/.

[4] Barrett, C. and C. Tinelli, *CVC3*, in: W. Damm and H. Hermanns, editors, *CAV*, LNCS **4590** (2007), pp. 298–302.

[5] Conchon, S., E. Contejean, J. Kanig and S. Lescuyer, *Lightweight Integration of the Ergo Theorem Prover inside a Proof Assistant*, in: *Second Automated Formal Methods workshop series (AFM07)*, Atlanta, Georgia, USA, 2007.

[6] de Moura, L. M. and N. Bjørner, *Efficient E-matching for SMT solvers*, in: F. Pfenning, editor, *CADE*, LNCS **4603** (2007), pp. 183–198.

[7] Detlefs, D., G. Nelson and J. B. Saxe, *Simplify: A theorem prover for program checking*, Technical Report HPL-2003-148, HP Labs (2003).

[8] Dutertre, B. and L. M. de Moura, *A fast linear-arithmetic solver for DPLL(T).*, in: T. Ball and R. B. Jones, editors, *CAV*, LNCS **4144** (2006), pp. 81–94.

[9] Flanagan, C., R. Joshi and J. B. Saxe, *An explicating theorem prover for quantified formulas*, Technical Report 199, HP Labs (2004).

[10] Flanagan, C., K. Leino, M. Lillibridge, G. Nelson, J. B. Saxe and R. Stata, *Extended static checking for Java*, in: *ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'2002)*, 2002, pp. 234–245.

[11] Kiniry, J. R. and D. R. Cok, *ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an Internet voting tally system*, in: *Construction and Analysis of Safe, Secure and Interoperable Smart Devices: International Workshop, CASSIS 2004*, LNCS **3362**, 2005.

[12] Kozen, D., *Complexity of finitely generated algebras*, in: *Proceedings of the* 9^{th} *Symposium on Theory of Computing*, 1977, pp. 164–177.

[13] Moskal, M., K. Skalski, P. Olszta et al., *Nemerle programming language*, http://nemerle.org/.

[14] Nieuwenhuis, R. and A. Oliveras, *Proof-Producing Congruence Closure*, in: J. Giesl, editor, *16th International Conference on Term Rewriting and Applications, RTA'05*, Lecture Notes in Computer Science **3467** (2005), pp. 453–468.

[15] Plotkin, G. D., *Building-in equational theories*, in: D. Michie and B. Meltzer, editors, *Machine Intelligence* (1972), pp. 73–90.

[16] Ramakrishnan, I. V., R. C. Sekar and A. Voronkov, *Term indexing.*, in: J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, Elsevier and MIT Press, 2001 pp. 1853–1964.

[17] Ranise, S. and C. Tinelli, *The Satisfiability Modulo Theories Library (SMT-LIB)*, http://www.smt-lib.org (2007).

A Detailed Experimental Results

The first column lists the benchmark name, the second, third and fourth columns are the average times spent matching a single trigger during proof search for a given benchmark. The times are given in microseconds. The second column refer to the subtrigger matcher (Fig. 4), the third one for the subtrigger matcher combined with the flat matcher (Fig. 5) and the last column refers to the Simplify matcher (Fig. 3).

Benchmark name	**Subtrig.**	**Subtrig. + Flat**	**Simpl.**
AssignToNonInvariantField.ClientClass..ctor	248	180	520
Assumptions.Sub..ctor	214	166	444
Branching.T.M.T.notnull.System.Int32	237	217	359
Chunker0.Chunker..ctor.System.String.System.Int32	133	102	250
DefaultLoopInv0.A.M-modifiesOnLoop-noinfer	173	146	293
Immutable.test3.C..ctor	232	177	549
Interval.Interval.Shift.System.Int32	261	216	564
ModifyOther.Test..ctor	246	179	438
PeerFields.Child..ctor.System.Int32	285	209	602
PeerFields.Parent.M	269	234	494
PeerFields.Parent.P	201	174	369
PeerFields.PeerFields.Assign1.PeerFields	176	140	258
PeerFields.PeerFields.M.System.Int32	284	258	470
PeerModifiesClauses.Homeboy.T-level.2	224	204	313
PureReceiverMightBeCommitted.C..ctor	199	156	351
PureReceiverMightBeCommitted.C.N	189	174	363
QuantifierVisibilityInvariant.A..ctor	245	251	524
QuantifierVisibilityInvariant.B..ctor.int	275	242	479
QuantifierVisibilityInvariant.C..ctor.int	280	233	482
Strengthen.MainClass.Main.HARD	190	163	458
Strings.test3.MyStrings.StringCoolness2.bool	184	153	418
Strings.test3.MyStrings.StringCoolness3	162	129	369
Types.T..ctor.D.notnull-orderStrength.1	142	83	428
ValidAndPrevalid.Interval.Foo4	251	194	545
Average	221	183	431

Benchmark name	Subtrig.	Subtrig. + Flat	Simpl.
javafe.CopyLoaded.019	322	262	826
javafe.ast.AmbiguousMethodInvocation.007	112	70	709
javafe.ast.AmbiguousVariableAccess.007	114	77	775
javafe.ast.ArrayRefExpr.007	106	67	726
javafe.ast.CastExpr.007	113	77	744
javafe.ast.ClassLiteral.007	113	70	764
javafe.ast.CondExpr.007	111	77	739
javafe.ast.FieldAccess.009	118	70	725
javafe.ast.InstanceOfExpr.007	112	72	726
javafe.ast.MethodInvocation.009	122	71	800
javafe.ast.NewArrayExpr.009	112	73	750
javafe.ast.NewInstanceExpr.003	540	272	3959
javafe.ast.NewInstanceExpr.008	108	72	709
javafe.ast.ParenExpr.007	113	76	758
javafe.ast.ThisExpr.007	115	75	723
javafe.ast.VariableAccess.008	109	74	714
javafe.parser.Lex.006	218	189	1056
javafe.parser.Lex.018	256	190	874
javafe.parser.Parse.005	275	202	1864
javafe.parser.Parse.006	354	296	2107
javafe.reader.ASTClassFileParser.005	730	373	5498
javafe.reader.ASTClassFileParser.022	690	415	4544
javafe.tc.Env.007	742	517	5980
javafe.tc.EnvForLocals.001	268	190	869
Average	249	164	1581