To Ackermann-ize or not to Ackermann-ize? On Efficiently Handling Uninterpreted Function Symbols in $SMT(\mathcal{EUF} \cup \mathcal{T})$ *

Roberto Bruttomesso[1], Alessandro Cimatti[1], Anders Franzén[1,2], Alberto Griggio[2], Alessandro Santuari[2], and Roberto Sebastiani[2]

[1] ITC-IRST, Povo, Trento, Italy. {bruttomesso,cimatti,franzen}@itc.it
[2] DIT, Università di Trento, Italy. {griggio,santuari,rseba}@dit.unitn.it

Abstract. Satisfiability Modulo Theories ($SMT(\mathcal{T})$) is the problem of deciding the satisfiability of a formula with respect to a given background theory $\mathcal{T}$. When $\mathcal{T}$ is the combination of two simpler theories $\mathcal{T}_1$ and $\mathcal{T}_2$ ($SMT(\mathcal{T}_1 \cup \mathcal{T}_2)$), a standard and general approach is to handle the integration of $\mathcal{T}_1$ and $\mathcal{T}_2$ by performing some form of search on the equalities between the shared variables.
A frequent and very relevant sub-case of $SMT(\mathcal{T}_1 \cup \mathcal{T}_2)$ is when $\mathcal{T}_1$ is the theory of Equality and Uninterpreted Functions ($\mathcal{EUF}$). For this case, an alternative approach is to eliminate first all uninterpreted function symbols by means of Ackermann's expansion, and then to solve the resulting *SMT* ($\mathcal{T}_2$) problem.
In this paper we build on the empirical observation that there is no absolute winner between these two alternative approaches, and that the performance gaps between them are often dramatic, in either direction.
We propose a simple technique for estimating a priori the costs and benefits, in terms of the size of the search space of an *SMT* tool, of applying Ackermann's expansion to all or part of the function symbols.
A thorough experimental analysis, including the benchmarks of the SMT'05 competition, shows that the proposed technique is extremely effective in improving the overall performance of the *SMT* tool.

1 Introduction

Satisfiability Modulo a Theory $\mathcal{T}$ ($SMT(\mathcal{T})$) is the problem of checking the satisfiability of a quantifier-free (or ground) first-order formula with respect to a given first-order theory $\mathcal{T}$ (we are considering theories with equality). Theories of interest for many applications are, e.g., the theory of difference logic $\mathcal{DL}$, the theory $\mathcal{EUF}$ of equality and uninterpreted functions, the quantifier-free fragment of Linear Arithmetic over the rationals $\mathcal{LA}(\mathbb{Q})$ and that over the integers $\mathcal{LA}(\mathbb{Z})$, the theory of bit-vectors $\mathcal{BV}$. The prominent *lazy* approach to $SMT(\mathcal{T})$, which underlies several systems (e.g., CVCLITE [3], DLSAT [10], DPLL(T)/BarceLogic [12], MATHSAT [5], TSAT++ [2], ICS/YICES [11]), is based on extensions of propositional SAT technology: a SAT

* This work has been partly supported by ISAAC, an European sponsored project, contract no. AST3-CT-2003-501848, by ORCHID, a project sponsored by Provincia Autonoma di Trento, and by a grant from Intel Corporation.

solver is modified to enumerate boolean assignments, and integrated with a decision procedure for sets of literals in the theory $\mathcal{T}$ ($\mathcal{T}$*-solver*).

When $\mathcal{T}$ is the combination of two simpler theories $\mathcal{T}_1$ and $\mathcal{T}_2$ (*SMT*$(\mathcal{T}_1 \cup \mathcal{T}_2)$), a standard and general approach is to handle the integration of $\mathcal{T}_1$ and $\mathcal{T}_2$ by performing some form of search on the equalities between the variables which are shared between the theories (*interface equalities*): in the Nelson-Oppen [13] and Shostak [15] schemata (NO hereafter), the interface equalities are deduced by the $\mathcal{T}$*-solver*s; in the Delayed Theory Combination schema (DTC hereafter) [6, 7] all or part of them are assigned to truth values also by the underlying SAT solver.

A frequent and very relevant sub-case is when one of the two theories is that of equality and uninterpreted functions $\mathcal{EUF}$. (Hereafter we refer to this problem as *SMT* ($\mathcal{EUF} \cup \mathcal{T}$).) For this case, an alternative approach is to eliminate first all uninterpreted function symbols by means of Ackermann's expansion [1], and then to solve the resulting single-theory *SMT*$(\mathcal{T})$ problem. (Hereafter we refer to this approach as ACK.)

In this paper we focus on *SMT* ($\mathcal{EUF} \cup \mathcal{T}$). Comparing the performances of DTC and ACK approaches, we notice that not only there is no absolute winner, but also the performance gaps are often dramatic, in either direction. We investigate the causes of this fact, and we introduce a technique for estimating off-line the costs and benefits, in terms of the size of the search space of an *SMT* tool, of applying Ackermann's expansion to all or part of the function symbols.

We have implemented a preprocessor which analyzes the input formula, decides autonomously which functions to expand, performs such expansions and gives the resulting formula as input to an *SMT* tool.

A thorough experimental analysis, including the benchmarks of the SMT'05 competition, shows that our preprocessor performs the best choice(s) nearly always, and that the proposed technique is extremely effective in improving the overall performance of the *SMT* tool.

The paper is organized as follows. In §2 we introduce the necessary background information on *SMT*, *SMT*$(\mathcal{T}_1 \cup \mathcal{T}_2)$, DTC and Ackermann's expansion. In §3 we present the main intuitions and ideas underlying our work. In §4 we present our new preprocessor. In §5 we present the experimental evaluation of our work. In §6 we conclude and briefly present potential future developments.

2 Background

2.1 Satisfiability Modulo Theory

Fig. 1 presents Bool$+\mathcal{T}$, (a much simplified version of) a standard schema of a decision procedure for *SMT*$(\mathcal{T})$. The function *Atoms*(φ) takes a ground formula φ and returns the set of atoms which occur in φ. We use the notation φ^p to denote the *propositional abstraction* of φ, which is formed by the function $\mathcal{T}2\mathcal{B}$ that maps propositional variables to themselves, ground atoms into fresh propositional variables, and is homomorphic w.r.t. boolean operators and set inclusion. The function $\mathcal{B}2\mathcal{T}$ is the inverse of $\mathcal{T}2\mathcal{B}$. We use μ^p to denote a propositional assignment, i.e. a conjunction (a set) of propositional literals. (If $\mathcal{T}2\mathcal{B}(\mu) \models \mathcal{T}2\mathcal{B}(\varphi)$, then we say that μ *propositionally satisfies* φ.)

```
function Bool+T (φ: quantifier-free formula)
1       A^p ⟵ T2B(Atoms(φ))
2       φ^p ⟵ T2B(φ)
3       while Bool-satisfiable(φ^p) do
4           μ^p ⟵ pick_total_assign(A^p, φ^p)
5           (ρ, π) ⟵ T − satisfiable(B2T(μ^p))
6           if ρ = sat then return sat
7           φ^p ⟵ φ^p ∧ ¬T2B(π)
8       end while
9       return unsat
end function
```

Fig. 1. A simplified view of enumeration-based T-satisfiability procedure: Bool+$\mathcal{T}$

The idea underlying the algorithm is that the truth assignments for the propositional abstraction of φ are enumerated and checked for satisfiability in $\mathcal{T}$. The procedure either returns sat if one such model is found, or returns unsat otherwise. The function *pick_total_assign* returns a total assignment to the propositional variables in φ^p, that is, it assigns a truth value to all variables in $\mathcal{A}^p$. The function $\mathcal{T}$-satisfiable(μ) detects if the set of conjuncts μ is $\mathcal{T}$-satisfiable: if so, it returns (sat, $\emptyset$); otherwise, it returns (unsat, π), where $\pi \subseteq \mu$ is a $\mathcal{T}$-unsatisfiable set, called a *theory conflict set*. We call the negation of a conflict set, a *conflict clause*.

The algorithm is a coarse abstraction of the ones underlying most *SMT* tools (including, e.g., TSAT++, MATHSAT, DLSAT, DPLL(T)/BarceLogic, CVCLITE, ICS/YICES).

In practice, the enumeration is carried out by means of efficient implementations of the DPLL algorithm [16], where a *partial assignment* μ^p is built incrementally, and *unit propagation* is used extensively to perform all the assignments which derive deterministically from the current μ^p. Conflict sets, generated because either the current μ^p falsifies the formula or because $\mathcal{T}$-satisfiable($\mathcal{B}2\mathcal{T}(\mu^p)$) fails, are used to prune the search tree and to backtrack as high as possible (*backjumping*), and *learned* as conflict clauses to avoid generating the same conflicts in future branches. Another important improvement is *early pruning*: intermediate assignments are checked for $\mathcal{T}$-satisfiability and, if not $\mathcal{T}$-satisfiable, then are pruned (since no refinement can be $\mathcal{T}$-satisfiable); finally, *theory deduction* can be used to reduce the search space by explicitly returning truth values for unassigned literals, as well as constructing/learning implications. The interested reader is pointed to [5, 8, 3, 12, 11] for details and further references.

2.2 $SMT(T_1 \cup T_2)$ via theory combination

In many practical applications of $SMT(\mathcal{T})$, the background theory is a combination of two (or more) theories $\mathcal{T}_1$ and $\mathcal{T}_2$. Most approaches to $SMT(\mathcal{T}_1 \cup \mathcal{T}_2)$ rely on the adaptation of the Bool+$\mathcal{T}$ schema, by instantiating $\mathcal{T}$-satisfiable with some decision procedure for the satisfiability of $\mathcal{T}_1 \cup \mathcal{T}_2$, typically based on an integration schema like Nelson-Oppen (NO) [13] (or its variant due to Shostak [15]), or on the more recent Delayed Theory Combination (DTC) schema [6, 7].

```
function DTC (φ_i: quantifier-free formula)
1    φ ⟵ purify(φ_i)
2    A^p ⟵ T2B(Atoms(φ) ∪ interface_equalities(φ))
3    φ^p ⟵ T2B(φ)
4    while Bool-satisfiable (φ^p) do
5        μ_1^p ∧ μ_2^p ∧ μ_e^p = μ^p ⟵ pick_total_assign(A^p, φ^p)
6        (ρ_1, π_1) ⟵ T_1-satisfiable (B2T(μ_1^p ∧ μ_e^p))
7        (ρ_2, π_2) ⟵ T_2-satisfiable (B2T(μ_2^p ∧ μ_e^p))
8        if (ρ_1 = sat ∧ ρ_2 = sat) then return sat else
9            if ρ_1 = unsat then φ^p ⟵ φ^p ∧ ¬T2B(π_1)
10           if ρ_2 = unsat then φ^p ⟵ φ^p ∧ ¬T2B(π_2)
11   end while
12   return unsat
end function
```

Fig. 2. A simplified view of the DTC procedure for $SMT(\mathcal{T}_1 \cup \mathcal{T}_2)$

Both the NO and DTC schemata work only for combinations of *stably-infinite* and *signature-disjoint* theories $\mathcal{T}_i$ with equality (we recall that $\mathcal{T}_i$ is stably-infinite iff every quantifier-free $\mathcal{T}_i$-satisfiable formula is satisfiable in an infinite model of $\mathcal{T}_i$). Moreover, they require the input formula to be *pure*: a formula φ is pure iff every atom ψ in φ is *i-pure* for some $i \in \{1,2\}$, that is ψ contains only $=$, variables and symbols from the signature of $\mathcal{T}_i$. Every non-pure $\mathcal{T}_1 \cup \mathcal{T}_2$ formula φ can be converted into an equivalently satisfiable pure formula φ' by recursively labeling terms t with fresh variables v_t, and by conjoining the definition atom $(v_t = t)$ to the formula. E.g.:

$$(f(x+3y) = g(2x-y)) \;\Rightarrow\; (f(v_{x+3y}) = g(v_{2x-y})) \wedge (v_{x+3y} = x+3y) \wedge (v_{2x-y} = 2x-y).$$

This process is called *purification*, and is linear in the size of the input formula.

In a pure formula φ, an *interface variable* is a variable appearing in both 1-pure and 2-pure atoms. An *interface equality* is an equality between two interface variables.

In the NO schema, the two decision procedures for $\mathcal{T}_1$ and $\mathcal{T}_2$ ($\mathcal{T}_i$-solvers) cooperate by exchanging (disjunctions of) interface equalities (e_{ij}'s). In the DTC schema, each of the two $\mathcal{T}_i$-solvers works in isolation, without direct exchange of information. Their mutual consistency is ensured by augmenting the input problem with all interface equalities e_{ij}, even if these do not occur in the original problem. The enumeration of assignments includes not only the atoms in the formula, but also the interface equalities e_{ij}. Both theory solvers receive, from the boolean level, the same truth assignment μ_e for e_{ij}: under such conditions, the two "partial" models found by each decision procedure can be merged into a model for the input formula.

A simplified view of the DTC algorithm is presented in Fig. 2. Initially (lines 1–3), the formula is purified, the e_{ij}'s which do not occur in the purified formula are created and added to the set of propositional symbols $\mathcal{A}^p$, and the propositional abstraction φ^p of φ is created. Then, the main loop is entered (lines 4–11): while φ^p is propositionally satisfiable (line 4), a satisfying truth assignment μ^p is selected (line 5). Truth values are associated not only to atoms in φ, but also to the e_{ij} atoms, even though they do

not occur in φ. μ^p is then (implicitly) separated into $\mu_1^p \wedge \mu_e^p \wedge \mu_2^p$, where $\mathcal{B}2\mathcal{T}(\mu_i^p)$ is a set of i-pure literals and $\mathcal{B}2\mathcal{T}(\mu_e^p)$ is a set of e_{ij}-literals. The relevant part of μ^p are checked for consistency against each theory (lines 6–7); $\mathcal{T}_i$-satisfiable(μ) returns a pair (ρ_i, π_i), where ρ_i is unsat iff μ is unsatisfiable in $\mathcal{T}_i$, and sat otherwise. If both calls to $\mathcal{T}_i$-satisfiable return sat, then the formula is satisfiable. Otherwise, when ρ_i is unsat, then π_i is a theory conflict set, i.e. $\pi_i \subseteq \mu$ and π_i is $\mathcal{T}_i$-unsatisfiable. Then, φ^p is strengthened to exclude truth assignments which may fail in the same way (line 9–10), and the loop is resumed. Unsatisfiability is returned (line 12) when the loop is exited without having found a model.

In practical implementations of DTC, as before, the enumeration is carried out by means of efficient implementations of the DPLL engine, where a *partial assignment* μ^p is built incrementally, exploiting unit-propagation, backjumping and learning, early pruning and theory deduction. Moreover, if one or both *T_i-satisfiable* have the capability of deducing (disjunctions of) interface equalities which derive from $\mathcal{T}_i$ from a partial assignment μ, [3] then such a deduction is exploited to prune the boolean search on the interface equalities (*e_{ij}-deduction*). To this extent, DTC extends the NO schema, in the sense that it allows for using *T_i-satisfiable* procedures with every deduction capability, trading e_{ij}-deduction power with boolean search, and allows for emulating the NO schema [9]. For the sake of simplicity, in this paper we do not consider e_{ij}-deduction for DTC. We refer the reader to [7, 9] for a more detailed discussion.

Example 1. Let φ be the following $\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Z})$-pure formula

$$\begin{aligned} \varphi \;\equiv\; & w = h(x) \wedge a = h(y) \wedge c = f(z) \wedge d = f(b) \wedge f(c) = f(b) \wedge \\ & w = f(d) \wedge \neg(c = d) \wedge x \geq y \wedge x \leq y \wedge z = w - a \wedge b = 0. \end{aligned} \tag{1}$$

x, y, z, w, a, b are the interface variables, so that there are 15 interface equalities: $z = b, w = b, a = b, x = b, y = b, z = w, z = a, x = z, y = z, w = a, x = w, y = w, x = a, y = a, x = y$.
The DPLL solver generates first the assignment $\mu := \mu_{\mathcal{EUF}} \cup \mu_{\mathcal{LA}(\mathbb{Z})}$ satisfying φ, s.t.
$\mu_{\mathcal{EUF}} := \{w = h(x), a = h(y), c = f(z), d = f(b), f(c) = f(b), w = f(d), \neg(c = d)\}$,
$\mu_{\mathcal{LA}(\mathbb{Z})} := \{x \geq y, x \leq y, z = w - a, b = 0\}$.
Then it tries to extend it with a total truth assignment μ_e to the interface equalities such that $\mu_{\mathcal{EUF}} \cup \mu_e$ and $\mu_{\mathcal{LA}(\mathbb{Z})} \cup \mu_e$ are consistent in $\mathcal{EUF}$ and $\mathcal{LA}(\mathbb{Z})$ respectively. This requires some search on the 15 interface equalities.

E.g, if the DPLL engine is smart or lucky enough to select first $x = y$, $w = a$, $z = b$, then we have
$\mu_{\mathcal{LA}(\mathbb{Z})} \cup \{\neg(x = y)\} \models_{\mathcal{LA}(\mathbb{Z})} \bot$, so that $x = y$ is added to μ,
$\mu_{\mathcal{EUF}} \cup \{x = y, \neg(w = a)\} \models_{\mathcal{EUF}} \bot$, so that $w = a$ is added to μ,
$\mu_{\mathcal{LA}(\mathbb{Z})} \cup \{x = y, w = a, \neg(z = b)\} \models_{\mathcal{LA}(\mathbb{Z})} \bot$, so that $z = b$ is added to μ,
$\mu_{\mathcal{EUF}} \cup \{x = y, w = a, z = b\} \models_{\mathcal{EUF}} \bot$, hence φ is $\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Z})$-inconsistent. □

Notice that on a single-theory $SMT(\mathcal{T})$ problem, DTC behaves as a standard *SMT* tool, because there are no interface equalities.

[3] In the NO schema this capability is strictly required for both *T_i-satisfiable*'s [13].

2.3 $SMT(\mathcal{EUF} \cup \mathcal{T})$ via Ackermann's expansion

When one of the theories $\mathcal{T}_i$ is $\mathcal{EUF}$, another possible approach to the $SMT(\mathcal{T}_1 \cup \mathcal{T}_2)$ problem is to eliminate uninterpreted function symbols by means of Ackermann's expansion [1] so to obtain an $SMT(\mathcal{T})$ problem with only one theory. The method works by replacing every function application occurring in the input formula φ with a fresh variable and then adding to φ all the needed functional consistency constraints. The new formula φ' obtained is equisatisfiable with φ, and contains no uninterpreted function symbols. First, each distinct function application $f(x_1,\ldots,x_n)$ is replaced by a fresh variable $v_{f(x_1,\ldots,x_n)}$. Then, for every pair of distinct applications of the same function, $f(x_1,\ldots,x_n)$ and $f(y_1,\ldots,y_n)$, a constraint

$$(\bigwedge_{i=1}^{arity(f)} ack(x_i) = ack(y_i)) \rightarrow v_{f(x_1,\ldots,x_n)} = v_{f(y_1,\ldots,y_n)}, \tag{2}$$

is added, where *ack* is a function that maps each function application $g(z_1,\ldots,z_n)$ into the corresponding variable $v_{g(z_1,\ldots,z_n)}$, each variable into itself and is homomorphic wrt. the interpreted symbols. The atom $ack(x_i) = ack(y_i)$ is not added if the two sides of the equality are syntactically identical.

Example 2. Let φ be the pure formula (1) of Example 1. Then, replacing every function application with a fresh variable, and adding all the functional consistency constraints, we obtain the formula

$$\begin{aligned}
\varphi_{\text{ACK}} \equiv\ & w = v_{h(x)} \wedge a = v_{h(y)} \wedge c = v_{f(z)} \wedge d = v_{f(b)} \wedge v_{f(c)} = v_{f(b)} \wedge \\
& w = v_{f(d)} \wedge \neg(c = d) \wedge x \geq y \wedge x \leq y \wedge z = w - a \wedge b = 0 \wedge \\
& (x = y \rightarrow v_{h(x)} = v_{h(y)}) \wedge (z = b \rightarrow v_{f(z)} = v_{f(b)}) \wedge \\
& (z = c \rightarrow v_{f(z)} = v_{f(c)}) \wedge (z = d \rightarrow v_{f(z)} = v_{f(d)}) \wedge \\
& (c = b \rightarrow v_{f(c)} = v_{f(b)}) \wedge (c = d \rightarrow v_{f(c)} = v_{f(d)}) \wedge \\
& (b = d \rightarrow v_{f(b)} = v_{f(d)}).
\end{aligned} \tag{3}$$

The DPLL solver first deterministically selects the truth assignment

$$\mu_{\mathcal{LA}(\mathbb{Z})} := \{\ w = v_{h(x)}, a = v_{h(y)}, c = v_{f(z)}, d = v_{f(b)}, v_{f(c)} = v_{f(b)}, w = v_{f(d)}, \\ \neg(c = d), x \geq y, x \leq y, z = w - a, b = 0\},$$

which is consistent in $\mathcal{LA}(\mathbb{Z})$. Then, it performs some search on the remaining 12 equalities. [4]

E.g., if it is smart or lucky enough to select first $x = y$, $z = b$, then we have:
$\mu_{\mathcal{LA}(\mathbb{Z})} \cup \{\neg(x = y)\} \models_{\mathcal{LA}(\mathbb{Z})} \bot$, so that $x = y$ is added to μ,
$\mu_{\mathcal{LA}(\mathbb{Z})} \cup \{x = y, v_{h(x)} = v_{h(y)}, \neg(z = b)\} \models_{\mathcal{LA}(\mathbb{Z})} \bot$, so that $z = b$ is added to μ,
$\mu_{\mathcal{LA}(\mathbb{Z})} \cup \{x = y, v_{h(x)} = v_{h(y)}, z = b, v_{f(z)} = v_{f(b)}\} \models_{\mathcal{LA}(\mathbb{Z})} \bot$, hence φ is $\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Z})$-inconsistent. □

[4] The remaining equalities are only 12 because $v_{f(c)} = v_{f(b)}$ causes the removal of the 5th implication.

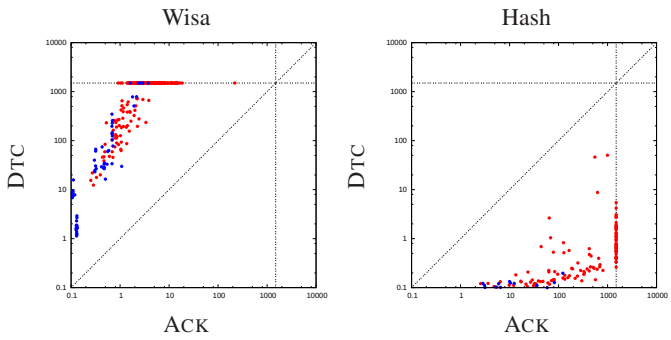


Fig. 3. Execution time ratio (in logarithmic scale) for DTC and ACK on the benchmarks Wisa and Hash, using MATHSAT. A dot above the diagonal line means better performance of ACK and vice versa. The horizontal and vertical dashed lines represent time-out.

Notice that, for simplicity, in Example 1 we have considered a pure formula φ, which might be the result of purifying some non-pure formula φ'. If so, applying Ackermann expansion directly to φ' might result into a more compact formula than (3).

Henceforth, we call respectively *Ackermann constraints* or *Ackermann implications* the functional consistency constraints added by Ackermann expansion, *Ackermann equalities* the equalities occurring in the Ackermann constraints, and *Ackermann variables* the variables occurring in the Ackermann equalities.

3 To Ackermann-ize or not to Ackermann-ize?

We start from a simple empirical observation: neither DTC or ACK always prevails in the task of solving $SMT(\mathcal{EUF} \cup \mathcal{T})$ problems, and the performance gaps between the two approaches may be dramatic, in either direction. As an example, Figure 3 shows the execution times of the two approaches on two different groups of benchmarks, for the MATHSAT [8] solver (both tests will be described in §5). For the Wisa benchmarks (left), ACK is up to 1000 times faster than DTC (or even more, considering also the timed-out examples) , whilst for the Hash benchmarks (right) the converse is true.

By tracing the behavior of MATHSAT on these tests, we notice that the performance gaps mirror the different amount of boolean search performed by the two techniques. From which we argue that one of the main reasons of such big performance gaps is the different size of the boolean search space that each technique has to explore in order to decide the satisfiability of its input.

Thus, we look to both techniques from the perspective of the boolean search only. Both DTC and ACK require the SAT solver to perform an extra boolean search on equalities which did not occur in the original formula (i.e., on the interface equalities and on the Ackermann equalities respectively). Thus the enlargement of the boolean search space with the two techniques depends directly on the number of these new equalities introduced.

3.1 Enlargement of the search space with DTC

In the DTC approach it may be necessary to assign a truth value to up to all the interface equalities. If φ is a pure $\mathcal{EUF} \cup \mathcal{T}$ formula, then the number of interface equalities is given by $|\mathcal{V}| \cdot (|\mathcal{V}| - 1)/2$, where $|\mathcal{V}|$ is the number of interface variables in φ. (Notice that this is an upper bound for the number of the *new* equalities introduced, since some of them might already appear in φ.) Thus, with DTC, the number of boolean atoms the SAT solver may have to explore is enlarged by a factor that is quadratic in the number of the interface variables.

Example 3. The formula φ of Example 1 has 6 interface variables, so that the number of atoms the SAT solver may have to explore is increased by $(6 \cdot 5)/2 = 15$ interface equalities, all of which are new. □

Notice that, in general, the input problem φ must be purified to be handled by DTC. The purification process adds a number of new variables and atoms that is linear in the size of φ. However, this does not cause an enlargement of the boolean search space, because all the atoms added are definitions of terms like $(v_t = t)$ and occur as unit clauses in the resulting formula, so that they are assigned a priori and deterministically to true by the SAT solver.

3.2 Enlargement of the search space with ACK

In the ACK approach, the increase in the boolean search space depends on the number of (new) equalities in the Ackermann constraints introduced.

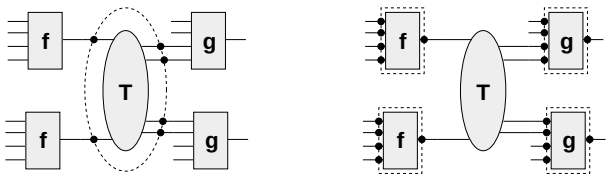
Let $\mathcal{F}$ be the set of (distinct) function symbols occurring in φ, and let O_f be the set of all (distinct) applications of the function f in the input formula φ. Then the number of new Ackermann equalities introduced is less than or equal to

$$\sum_{f \in \mathcal{F}} \frac{|O_f| \cdot (|O_f| - 1)}{2} \cdot (arity(f) + 1). \tag{4}$$

In fact, for each $f \in \mathcal{F}$ and for each of the $(|O_f| \cdot (|O_f| - 1))/2$ pairs of distinct occurrences of f, Equation (2) causes the introduction of up to $(arity(f) + 1)$ new Ackermann equalities. (As with DTC, this is an upper bound, both because some of the equalities in one constraint could already occur in the formula or in other constraints, and because identities like $x = x$ are dropped by construction.)

Thus, with ACK, the number of boolean atoms the SAT solver may have to explore is enlarged by a factor that is quadratic in the number of occurrences of each function symbol, and linear in the number of distinct function symbols and in their arity.

Example 4. In the formula φ (1) of Example 1, $O_h = 1$ and $O_f = 4$. Thus the Ackermann constraints introduced in the formula φ_{ACK} (3) of Example 2 contain $(2 \cdot 1)/2 \cdot (1 + 1) + (4 \cdot 3)/2 \cdot (1 + 1) = 14$ equalities. Since $v_{f(c)} = v_{f(b)}$ is not new, the new equalities are 13. Notice that also $c = b$ does not really increase the boolean search space, because the 5th implication is immediately removed by the DPLL solver (Footnote 4). □



(a) DTC (b) ACK

Fig. 4. Schemas of the frontier between $\mathcal{EUF}$ and $\mathcal{T}$ in the DTC and ACK approaches.

3.3 Intuition: the "frontier" between $\mathcal{EUF}$ and $\mathcal{T}$ in DTC and ACK

Both DTC and ACK introduce an enlargement of the search space of the input problem φ. Intuitively, we can think of this extra boolean search as the *cost* associated to each of the two approaches for handling the interaction between the two theories. We notice that the set of new equivalences introduced by either approach corresponds to a distinct notion of "frontier" between $\mathcal{EUF}$ and $\mathcal{T}$ in the two approaches.

In DTC, the frontier is given by the interface variables (see Figure 4.a). As the cost of DTC depends quadratically on the size of the frontier, DTC is expected to perform better for those examples where the two theories are loosely coupled, and worse when there is a strong connection between them.

With ACK, the frontier between the two theories is potentially much larger, because it consists of the inputs and outputs of all (distinct) function applications (i.e, the Ackermann variables), including those which do not interact with terms of the theory $\mathcal{T}$ (see Figure 4.b). However, in this case the cost is not quadratic in the number of variables in the frontier; rather, it depends on the number of different functions and of distinct occurrences of each function invocation (4). Thus ACK is expected to perform better when the number of the distinct function invocations for the same function is low.

4 Cost-driven Ackermann-ization

When we want to check the satisfiability of an $SMT(\mathcal{EUF} \cup \mathcal{T})$ formula φ, no matter which of the two approaches (DTC or ACK) we use, we must pay a price in terms of enlargement of the boolean search space. We believe that this cost is one of the main factors which influence the performance of the two methods. Thus, being able to estimate this cost a priori can drive the choice of which technique to apply.

4.1 A global-decision approach: DECIDE

Our first, basic idea is that of trying to estimate a priori the difference of costs of applying ACK or DTC, and to simply select the technique that costs less. We call this first idea "a global-decision approach" because here the decision involves all function symbols altogether.

The resulting algorithm DECIDE is outlined in Figure 5. Let φ be a (possibly non-pure) $SMT(\mathcal{EUF} \cup \mathcal{T})$ formula. The function *countAckEqualities* returns the number

```
function DECIDE (φ: quantifier-free formula)
1    ack_eq ⟵ countAckEqualities(φ)
2    int_eq ⟵ countInterfaceEqualities(φ)
3    if ack_eq < int_eq then return ackermanize(φ)
4    else return φ
end function
```

Fig. 5. High-level description of the DECIDE algorithm

of new Ackermann equalities added by the Ackermann's expansion of φ. The function *countInterfaceEqualities* returns the number of new interface equalities in (the formula resulting from purifying) φ. Notice that both functions return the *exact* number of equalities introduced, avoiding counting repeated equalities, identities, etc. Both functions are straightforward to implement, and their complexity is linear in the size of φ.

DECIDE works as a preprocessor for an *SMT* solver for $SMT(\mathcal{EUF} \cup \mathcal{T})$ which uses DTC: the algorithm either returns an Ackermann-ized version of the input φ (if ACK costs less), or leaves the input untouched. As noticed in §2, in the first case DTC behaves as a standard single-theory *SMT* tool, so that the two options correspond to ACK and DTC respectively.

Example 5. Consider again the formulas (1) and (3) of Examples 1 and 2 respectively. DTC would introduce 15 new interface equalities, whilst ACK would introduce 13 new Ackermann equalities. Therefore DECIDE in this case would choose ACK. □

4.2 A local-decision approach: PARTIAL

The idea just described can be generalized in the following way. From §3 we know that the cost of DTC depends quadratically on the global number of interface variables, whilst the cost of ACK, *for each function symbol* f, depends quadratically on the number of the distinct occurrences of f and linearly on its arity. Thus, we can decide to apply Ackermann's expansions only to *subsets* of the function symbols, according to their relative costs. We call this second idea "a local-decision approach" because here the decision involves subsets of function symbols.

Let f be a function in φ with very few occurrences but many arguments shared between $\mathcal{EUF}$ and $\mathcal{T}$. Then f causes a low increase of the ACK costs and a big increase of the DTC costs, because Ackermann's expansion will introduce few constraints, whilst the high number of interface variables would make DTC generate many new equalities. On the other hand, a function g with many occurrences but few or no arguments shared among the theories is going to cost much less for DTC than for ACK for the very same reason. Thus, if we consider a formula which contains both f and g, then applying Ackermann's expansion only *partially*, so that to remove only f, and solving the resulting problem with DTC, is going to cost less than pure ACK or pure DTC.

Example 6. Consider again the formula (1) of Example 1. If we expand only h, we get the following formula:

$$\begin{aligned} \varphi' \;\equiv\; & w = v_{h(x)} \wedge c = f(z) \wedge d = f(b) \wedge f(c) = f(b) \wedge w = f(d) \wedge \neg(c = d) \wedge \\ & x \geq y \wedge x \leq y \wedge z = w - v_{h(y)} \wedge b = 0 \wedge x = y \rightarrow v_{h(x)} = v_{h(y)}, \end{aligned} \tag{5}$$

```
function PARTIAL (φ: quantifier-free formula)
1       A ⟵ ∅
2       ψ ⟵ purify(φ)
3       do
4           B ⟵ selectFunctionsToAckermanize(ψ)
5           ψ ⟵ ackermanizeFunctions(ψ, B)
6           A ⟵ A ∪ B
7       while B ≠ ∅
8       φ' ⟵ ackermanizeFunctions(φ, A)
9       return φ'
end function
```

Fig. 6. High-level description of the PARTIAL algorithm

which has only 3 interface variables (z, b and w). Using DTC on φ' would then enlarge the search space by 3 interface equalities. Therefore, the mixed approach would cost in total 5 new equalities (2 for the Ackermann constraints and 3 for the interface equalities), which is less than with ACK (13) and DTC (15). □

The ideal solution would be to develop an algorithm that applies Ackermann's expansion to the subset of the function symbols corresponding to a global minimum in the number of new equalities to add. Unfortunately, finding such a global optimal solution seems to be very expensive. Intuitively, this is because both the cost and the benefit of applying Ackermann's expansion to each function symbol —in terms of more Ackermann equalities and less interface equalities to add respectively— depend on the previous eliminations of some other functions. (For example, as a consequence of the elimination of a function f, it may become convenient to eliminate also g because they had many pairs of corresponding arguments in common.) Thus, finding the global optimum may require exploring up to all the $2^{|\mathcal{F}|}$ possible subsets of function symbols.

For this reason, we have conceived instead the algorithm PARTIAL (outlined in Figure 6) which finds a local optimum. PARTIAL is a greedy algorithm that starts from the purified formula and that finds at each step a set of function symbols $\mathcal{B}$ whose removal causes a reduction in the number of equivalences to add. When this set is empty, a local minimum has been reached, and the algorithm terminates. Then the Ackermann's expansion on the set of selected functions $\mathcal{A}$ is performed on the original input formula φ, and the result is returned.

The core of PARTIAL is the function *selectFunctionsToAckermanize*, which returns the set of functions to remove in order to reduce the number of new equalities to add, according to the following heuristic. The function symbols occurring in φ are divided into (possibly overlapping) subgroups $\mathcal{G}_v$'s, one for every interface variable v in φ, $\mathcal{G}_v$ consisting of the set of all the function symbols that cause v to be an interface variable. Then the group $\mathcal{G}_v$ is returned which causes the maximum reduction $\mathsf{gain}_{\mathcal{G}_v}$ in terms of equivalences to add. (That is, $\mathsf{gain}_{\mathcal{G}_v}$ is defined as the difference between the number of interface equalities to remove and the number of equalities in the functional consistency constraints to add, if all the functions in the group were removed with

Ackermann's expansion.) If for no group $\mathcal{G}_v$ the value $\mathsf{gain}_{\mathcal{G}_v}$ is positive, then the empty set is returned. [5]

Example 7. Consider the pure formula (1) used in all the previous examples. When invoked for the first time, *selectFunctionsToAckermanize* constructs for the set of functions $\{f,h\}$ in (1) six groups, one for each interface variable:

$$\mathcal{G}_x = \mathcal{G}_y = \mathcal{G}_a = \{h\} \qquad\qquad \mathcal{G}_w = \mathcal{G}_z = \mathcal{G}_b = \{f\}.$$

Then, for each of them, the associated gain (i.e. the difference between the number of interface equalities to remove and the number of equalities to add for the functional consistency constraints) is computed:

$$\mathsf{gain}_{\mathcal{G}_x} = \mathsf{gain}_{\mathcal{G}_y} = \mathsf{gain}_{\mathcal{G}_a} = 12-2 = 10,\ \mathsf{gain}_{\mathcal{G}_w} = \mathsf{gain}_{\mathcal{G}_z} = \mathsf{gain}_{\mathcal{G}_b} = 12-11 = 1$$

because removing h makes x, y and a loose the status of interface variables, whilst removing f the same happens for w, z and b. Thus *selectFunctionsToAckermanize* selects $\{h\}$ only, causing the generation of the formula (5) of Example 6. At the next iteration of the main loop of PARTIAL, the only function symbol is f, which is not removed since all $\mathsf{gain}_{\mathcal{G}_v}$'s are negative. □

5 Empirical Evaluation

We implemented both DECIDE and PARTIAL in a preprocessor program, written in C++. It handles $SMT(\mathcal{EUF} \cup \mathcal{LA})$ problems, and has four different operational modes:

transparent (DTC), which simply reads a problem from its standard input and outputs it to its standard output without doing anything;
ackermanize, which removes every uninterpreted function symbol;
decide, which applies the DECIDE algorithm; and
partial, which applies the PARTIAL algorithm to remove a subset of the uninterpreted function symbols.

We tested our preprocessor with the MATHSAT [8] solver, which handles $SMT(\mathcal{EUF} \cup \mathcal{LA})$ problems with DTC. We used different benchmarks, coming from different domains:

QF_UFIDL comes from the SMT-LIB [14], and is made of formulas with $\mathcal{EUF}$ and integer difference logic. It is a superset of the QF_UFIDL set used in the SMT-COMP'05 competition [4];
Wisa are software verification benchmarks from the Wisconsin Safety Analyzer, created with a slightly modified version of the generator available at `http://www.cs.wisc.edu/wisa/papers/icse05/wisa-benchmarks.html`;

[5] As a direct consequence of how the groups are built, removing the functions in a group removes at least one interface variable from $\mathcal{V}$, so that at least $|\mathcal{V}|-1$ interface equalities are removed. It may be the case that more than one interface variable is removed: e.g., if $\mathcal{G}_x \subseteq \mathcal{G}_y$, then removing all the function symbols in $\mathcal{G}_y$ causes the removal of both x and y from $\mathcal{V}$.

EufLaArithmetic are simulations of arithmetic operations (succ, pred, sum) modulo N, using $\mathcal{EUF}$ and $\mathcal{LA}(\mathbb{Z})$. This and the following groups of benchmarks were introduced in [7];

Hash are problems over hash tables, modeled with a combination of $\mathcal{EUF}$ and $\mathcal{LA}(\mathbb{Z})$;

RandomCoupled are randomly generated $SMT(\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Q}))$ problems, with a propositional 3-CNF structure. In this group, there is a high coupling between the two theories, that is there is a high probability that for instance an argument of a function is a $\mathcal{LA}(\mathbb{Q})$ term;

RandomDecoupled are tests generated in the same way as the previous group, but where the coupling between $\mathcal{EUF}$ and $\mathcal{LA}(\mathbb{Q})$ is low.

The tests were run on a machine with an Intel Xeon 3GHz processor running Linux. The memory limit was set to 1GB, and the time limit to 1000 sec.

Figure 7 shows the results, both for the individual suites singularly and for the union of all the suites. A point in $\langle X, Y \rangle$ states that X problems have been solved each in less than or equal to Y seconds. (Notice the logarithmic scale of the Y axis.) A higher number of tests solved means better performance. When this number is the same, the lowest line is the best. All the plots include the cost of preprocessing, which however is negligible.

The following table summarizes the total results. The rows are sorted from the worst to the best, while the columns show details of the performances in terms of total number of tests solved, total running time, and total time to solve a fixed amount N of tests, for various values of N.

	Number of	**Total time**	**Total time for solving N tests**					
	tests solved	**(for all tests)**	**300**	**600**	**1200**	**1384**	**1479**	**1513**
transparent (DTC)	1384	34500	25.9	100.8	1804	34500	-	-
ackermanize	1479	41431	33.0	149.3	1402	5436	41431	-
decide	1513	12891	22.1	82.4	629	1646	3577	12891
partial	1516	13393	21.1	75.9	602	1495	3450	11335

We can see from both Figure 7 and the above table that different suites show very different performance gaps between **transparent** (DTC) and **ackermanize** (ACK), as observed in §2, and that both **decide** (DECIDE) and **partial** (PARTIAL) always behave quite similarly to the best of the two. (E.g., looking at the data, we noticed that **decide** chooses the most efficient option nearly always, and that the few samples for which it does not are such that the performance gaps between ACK and DTC are minor.)

The overall result shows that both DECIDE and PARTIAL are globally much more efficient than both ACK and DTC, with PARTIAL being the best technique. The reason why the performances of two techniques are so similar is that, on these benchmarks, it turns out that PARTIAL either removes all or most of the functions or it removes none, thus behaving very similarly to DECIDE.

6 Conclusions

In this paper we have focused on the $SMT(\mathcal{EUF} \cup \mathcal{T})$ problem. We have proposed a simple technique for estimating a priori the costs and benefits, in terms of the size of

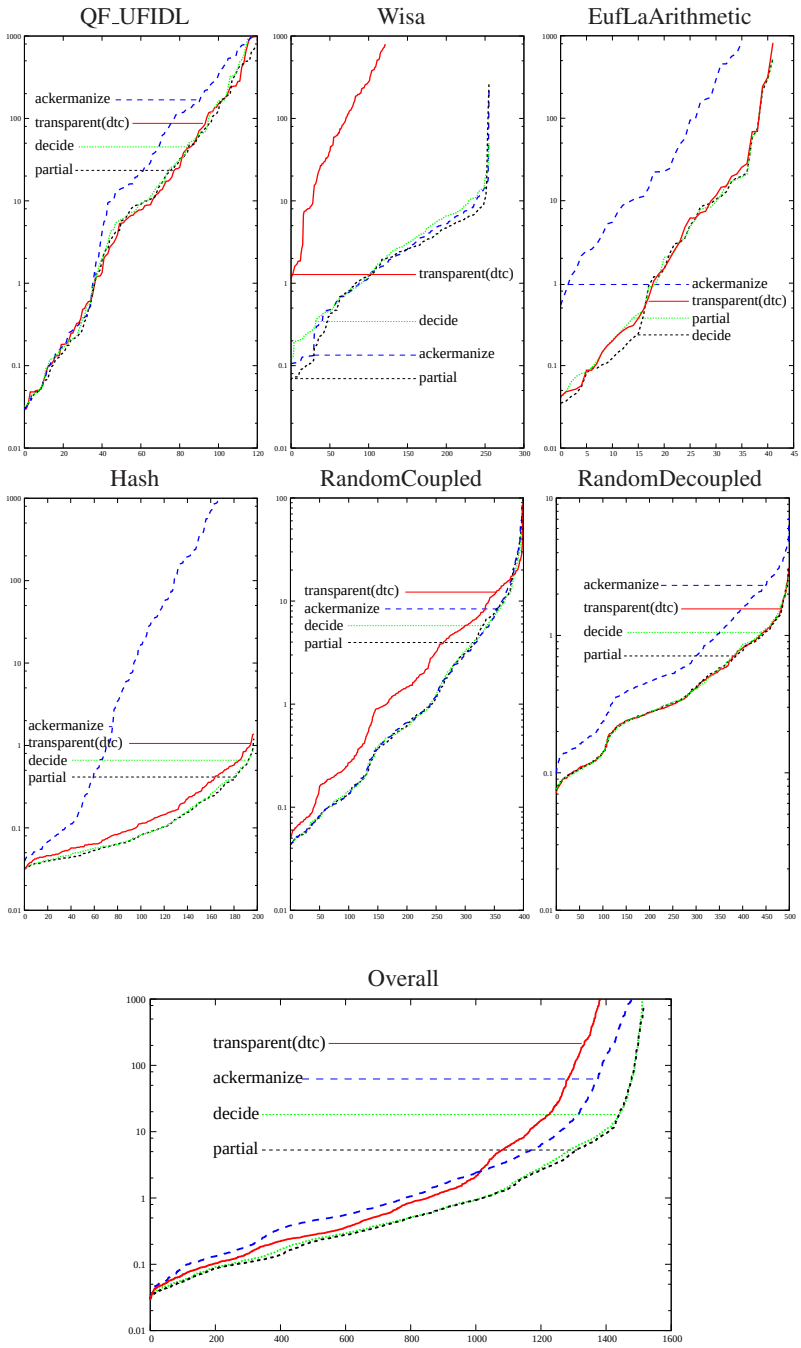


Fig. 7. Results of the benchmarks for the MATHSAT solver. For each technique, the X axis represents the number of tests solved and the Y axis the time required (in log scale). The labels in the plots are sorted according to performance: from the worst to the best.

the search space of an *SMT* tool, of applying Ackermann's expansion to all or part of the function symbols; we have implemented a preprocessor which analyzes the input formula, decides autonomously which functions to expand, performs such expansions and gives the resulting formula as input to an *SMT* tool; we have performed a thorough experimental analysis with MATHSAT on $SMT(\mathcal{EUF} \cup \mathcal{DL})$, $SMT(\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Q}))$ and $SMT(\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Z}))$, showing that the proposed technique is extremely effective in improving the overall performance of the *SMT* tool.

As future developments, we plan to experiment the effectiveness of our techniques also with other *SMT* tools (e.g., CVCLITE [3], ICS/YICES [11]), and with other theories (e.g., $\mathcal{EUF}$ with the theory of bit-vectors $\mathcal{BV}$).

References

1. W. Ackermann. Solvable Cases of the Decision Problem. North Holland Pub. Co., 1954.
2. A. Armando, C. Castellini, E. Giunchiglia, and M. Maratea. A SAT-based Decision Procedure for the Boolean Combination of Difference Constraints. In *Proc. SAT'04*, 2004.
3. C.L. Barrett and S. Berezin. CVC Lite: A New Implementation of the Cooperating Validity Checker. In *Proc. CAV'04*, volume 3114 of *LNCS*. Springer, 2004.
4. C. Barrett, L. de Moura, and A. Stump. SMT-COMP: Satisfiability Modulo Theories Competition. In *Proc. CAV'05*, volume 3576 of *LNCS*. Springer, 2005.
5. M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Schulz, and R. Sebastiani. An incremental and Layered Procedure for the Satisfiability of Linear Arithmetic Logic. In *Proc. TACAS'05*, volume 3440 of *LNCS*. Springer, 2005.
6. M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Ranise, and R. Sebastiani. Efficient Satisfiability Modulo Theories via Delayed Theory Combination. In *Proc. Int. Conf. on Computer-Aided Verification, CAV 2005.*, volume 3576 of *LNCS*. Springer, 2005.
7. M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Ranise, and R. Sebastiani. Efficient Theory Combination via Boolean Search. *Information and Computation*, 2005. To appear.
8. M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Schulz, and R. Sebastiani. MathSAT: A Tight Integration of SAT and Mathematical Decision Procedure. *Journal of Automated Reasoning*, 2005. to appear.
9. R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. Delayed Theory Combination vs. Nelson-Oppen for Satisfiability Modulo Theories: a Comparative Analysis. In *Proc. LPAR'06*, 2006.
10. S. Cotton, E. Asarin, O. Maler, and P. Niebert. Some Progress in Satisfiability Checking for Difference Logic. In *Proc. FORMATS-FTRTFT 2004*, 2004.
11. J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated Canonizer and Solver. In *Proc. CAV'01*, volume 2102 of *LNCS*, pages 246–249, 2001.
12. H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In *Proc. CAV'04*, volume 3114 of *LNCS*, pages 175–188. Springer, 2004.
13. G. Nelson and D.C. Oppen. Simplification by Cooperating Decision Procedures. *ACM Trans. on Programming Languages and Systems*, 1(2):245–257, 1979.
14. S. Ranise and C. Tinelli. The SMT-LIB Standard: Version 1.1. Technical Report, 2005.
15. R.E. Shostak. Deciding Combinations of Theories. *Journal of the ACM*, 31:1–12, 1984.
16. L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *Proc. CAV'02*, number 2404 in LNCS, pages 17–36. Springer, 2002.