Exploiting symmetry in SMT problems

David Déharbe[1], Pascal Fontaine[2],
Stephan Merz[2], and Bruno Woltzenlogel Paleo[3⋆]

[1] Universidade Federal do Rio Grande do Norte, Natal, RN, Brazil
david@dimap.ufrn.br
[2] University of Nancy and INRIA, Nancy, France
{Pascal.Fontaine,Stephan.Merz}@inria.fr
[3] Technische Universität Wien
bruno.wp@gmail.com

Abstract. Methods exploiting problem symmetries have been very successful in several areas including constraint programming and SAT solving. We here recast a technique to enhance the performance of SMT-solvers by detecting symmetries in the input formulas and use them to prune the search space of the SMT algorithm. This technique is based on the concept of (syntactic) invariance by permutation of constants. An algorithm for solving SMT by taking advantage of such symmetries is presented. The implementation of this algorithm in the SMT-solver veriT is used to illustrate the practical benefits of this approach. It results in a significant improvement of veriT's performances on the SMT-LIB benchmarks that places it ahead of the winners of the last editions of the SMT-COMP contest in the QF_UF category.

1 Introduction

While the benefit of symmetries has been recognized for the satisfiability problem of propositional logic [15], for constraint programming [9], and for finite model finding [4, 7, 11], SMT solvers (see [3] for a detailed account of techniques used in SMT solvers) do not yet fully exploit symmetries. Audemard et al. [1] use symmetries as a simplification technique for SMT-based model-checking, and the SMT solver HTP [14] uses some symmetry-based heuristics, but current state-of-the-art solvers do not exploit symmetries to decrease the size of the search space.

In the context of SMT solving, a frequent source of symmetries is when some terms take their value in a given finite set of totally symmetric elements. The idea here is very simple: given a formula G invariant by all permutations of some uninterpreted constants $c_0, \ldots, c_n$, for any model $\mathcal{M}$ of G, if term t does not contain these constants and $\mathcal{M}$ satisfies $t = c_i$ for some i, then there should be a model in which t equals c_0. While checking for unsatisfiability, it is thus sufficient to look for models assigning t and c_0 to the same value. This simple idea is very

⋆ This work was partly supported by the ANR DeCert project and the INRIA-CNPq project SMT-SAVeS.

effective, especially for formulas generated by finite instantiations of quantified problems. We have implemented our technique in a moderately efficient SMT solver (veriT [5]), and with this addition it outperforms the winners of recent editions of the SMT-COMP [2] contest in the QF_UF category. This indicates that detecting symmetries, automatically or based on hints in the input, can be important for provers to reduce the search space that they have to consider, just as some constraint solvers already take symmetry information into account.

Outline. We first introduce notations, then define symmetries and give the main theorem that allows us to reduce the search space. We recast an algorithm to exploit such symmetries in the context of SMT-solvers. Next, the classical pigeonhole problem is analyzed from the perspective of symmetries. Finally, some experimental results, based on the SMT-LIB, are provided and discussed.

2 Notations

A many-sorted first-order language is a tuple $\mathcal{L} = \langle \mathcal{S}, \mathcal{V}, \mathcal{F}, \mathcal{P}, d \rangle$ such that $\mathcal{S}$ is a countable non-empty set of disjoint sorts (or types), $\mathcal{V}$ is the (countable) union of disjoint countable sets $\mathcal{V}_\tau$ of variables of sort τ, $\mathcal{F}$ is a countably infinite set of function symbols, $\mathcal{P}$ is a countably infinite set of predicate symbols, and d assigns a sort in $\mathcal{S}^+$ to each function symbol $f \in \mathcal{F}$ and a sort in $\mathcal{S}^*$ to each predicate symbol $p \in \mathcal{P}$. Nullary predicates are propositions, and nullary functions are constants. The set of predicate symbols is assumed to contain a binary predicate $=_\tau$ for every sort $\tau \in \mathcal{S}$; since the sort of the equality can be deduced from the sort of the arguments, the symbol $=$ will be used for equality of all sorts. Terms and formulas over the language $\mathcal{L}$ are defined in the usual way.

An interpretation for a first-order language $\mathcal{L}$ is a pair $\mathcal{I} = \langle D, I \rangle$ where D assigns a non-empty domain D_τ to each sort $\tau \in \mathcal{S}$ and I assigns a meaning to each variable, function, and predicate symbol. As usual, the identity is assigned to the equality symbol. By extension, an interpretation $\mathcal{I}$ defines a value $\mathcal{I}[t]$ in D_τ for every term t of sort τ, and a truth value $\mathcal{I}[\varphi]$ in $\{\top, \bot\}$ for every formula φ. A model of a formula φ is an interpretation $\mathcal{I}$ such that $\mathcal{I}[\varphi] = \top$. The notation $\mathcal{I}_{s_1/r_1,\ldots,s_n/r_n}$ stands for the interpretation that agrees with $\mathcal{I}$, except that it associates the elements r_i of appropriate sort to the symbols s_i.

For convenience, we will consider that a theory is a set of interpretations for a given many-sorted language. The theory corresponding to a set of first-order axioms is thus naturally the set of models of the axioms. A theory may leave some predicates and functions uninterpreted: a predicate symbol p (or a function symbol f) is uninterpreted in a theory $\mathcal{T}$ if for every interpretation $\mathcal{I}$ in $\mathcal{T}$ and for every predicate q (resp., function g) of suitable sort, $\mathcal{I}_{p/q}$ belongs to $\mathcal{T}$ (resp., $\mathcal{I}_{f/g} \in \mathcal{T}$). It is assumed that variables are always uninterpreted in any theory, with a meaning similar to uninterpreted constants. Given a theory $\mathcal{T}$, a formula φ is $\mathcal{T}$-satisfiable if it has a model in $\mathcal{T}$. Two formulas are $\mathcal{T}$-equisatisfiable if one formula is $\mathcal{T}$-satisfiable if and only if the other is. A formula φ is a logical consequence of a theory $\mathcal{T}$ (noted $\mathcal{T} \models \varphi$) if every interpretation in $\mathcal{T}$ is a model

of φ. A formula φ is a $\mathcal{T}$-logical consequence of a formula ψ, if every model $\mathcal{M} \in \mathcal{T}$ of ψ is also a model of φ; this is noted $\psi \models_{\mathcal{T}} \varphi$. Two formulas ψ and φ are $\mathcal{T}$-logically equivalent if they have the same models in $\mathcal{T}$.

3 Defining symmetries

We now formally introduce the concept of formulas invariant w.r.t. permutations of uninterpreted symbols and study the $\mathcal{T}$-satisfiability problem of such formulas. Intuitively, the formula φ is invariant w.r.t. permutations of uninterpreted symbols if, modulo some syntactic normalization, it is left unchanged when the symbols are permuted. Formally, the notion of permutation operators depends on the theory $\mathcal{T}$ for which $\mathcal{T}$-satisfiability is considered, because only uninterpreted symbols may be permuted.

Definition 1. *A* permutation operator P *on a set* $\mathcal{R} \subseteq \mathcal{F} \cup \mathcal{P}$ *of uninterpreted symbols of a language* $\mathcal{L} = \langle \mathcal{S}, \mathcal{V}, \mathcal{F}, \mathcal{P}, d \rangle$ *is a sort-preserving bijective map from* $\mathcal{R}$ *to* $\mathcal{R}$*, that is, for each symbol* $s \in \mathcal{R}$*, the sorts of* s *and* $P[s]$ *are equal. A permutation operator homomorphically extends to an operator on terms and formulas on the language* $\mathcal{L}$*.*

As an example, a permutation operator on a language containing the three constants c_0, c_1, c_2 of identical sort, may map c_0 to c_1, c_1 to c_2 and c_2 to c_0.

To formally define that a formula is invariant by a permutation operator *modulo some rewriting*, the concept of $\mathcal{T}$-preserving rewriting operator is introduced.

Definition 2. *A* $\mathcal{T}$-preserving rewriting operator R *is any transformation operator on terms and formulas such that* $\mathcal{T} \models t = R[t]$ *for any term, and* $\mathcal{T} \models \varphi \Leftrightarrow R[\varphi]$ *for any formula* φ*. Moreover, for any permutation operator* P*, for any term and any formula,* $R \circ P \circ R$ *and* $R \circ P$ *should yield identical results.*

The last condition of Def. 2 will be useful in Lemma 6. Notice that R must be idempotent, since $R \circ P \circ R$ and $P \circ R$ should be equal for all permutation operators, including the identity permutation operator.

To better motivate the notion of a $\mathcal{T}$-preserving rewriting operator, consider a formula containing a clause $t = c_0 \vee t = c_1$. Obviously this clause is symmetric if t does not contain the constants c_0 and c_1. However, a permutation operator on the constants c_0 and c_1 would rewrite the formula into $t = c_1 \vee t = c_0$, which is not syntactically equal to the original one. Assuming the existence of some ordering on terms and formulas, a typical $\mathcal{T}$-preserving rewriting operator would reorder arguments of all commutative symbols according to this ordering. With appropriate data structures to represent terms and formulas, it is possible to build an implementation of this $\mathcal{T}$-preserving rewriting operator that runs in linear time with respect to the size of the DAG or tree that represents the formula.

Definition 3. *Given a $\mathcal{T}$-preserving rewriting operator R, a permutation operator P on a language $\mathcal{L}$ is a* symmetry operator *of a formula φ (a term t) on the language $\mathcal{L}$ w.r.t. R if $R[P[\varphi]]$ and $R[\varphi]$ (resp., $R[P[t]]$ and $R[t]$) are identical.*

Notice that, given a permutation operator P and a linear time $\mathcal{T}$-preserving rewriting operator R satisfying the condition of Def. 3, it is again possible to check in linear time if P is a symmetry operator of a formula w.r.t. R. In the following, we will assume a fixed rewriting operator R and say that P is a symmetry operator if it is a symmetry operator w.r.t. R.

Symmetries could alternatively be defined semantically, stating that a permutation operator P is a symmetry operator if $P[\varphi]$ is $\mathcal{T}$-logically equivalent to φ. The above syntactical symmetry implies of course the semantical symmetry. But the problem of checking if a permutation operator is a semantical symmetry operator has the same complexity as the problem of unsatisfiability checking. Indeed, consider the permutation P such that $P[c_0] = c_1$ and $P[c_1] = c_0$, and a formula ψ defined as $c = c_0 \wedge c \neq c_1 \wedge \psi'$ (where c, c_0 and c_1 do not occur in ψ'). To check if the permutation operator P is a semantical symmetry operator of ψ, it is necessary to check if formulas ψ and $P[\psi]$ are logically equivalent, which is only the case if ψ' is unsatisfiable.

Definition 4. *A term t (a formula φ) is* invariant w.r.t. permutations of uninterpreted constants $c_0, \ldots, c_n$ *if any permutation operator P on $c_0, \ldots, c_n$ is a symmetry operator of t (resp. φ).*

The main theorem follows: it allows one to introduce a symmetry breaking assumption in a formula that is invariant w.r.t. permutations of constants. This assumption will decrease the size of the search space.

Theorem 5. *Consider a theory $\mathcal{T}$, uninterpreted constants $c_0, \ldots, c_n$, a formula φ that is invariant w.r.t. permutations of $c_i, \ldots, c_n$, and a term t that is invariant w.r.t. permutations of $c_i, \ldots, c_n$. If $\varphi \models_{\mathcal{T}} t = c_0 \vee \ldots \vee t = c_n$, then φ is $\mathcal{T}$-satisfiable if and only if*

$$\varphi' =_{\text{def}} \varphi \wedge (t = c_0 \vee \ldots \vee t = c_i)$$

is also $\mathcal{T}$-satisfiable. Clearly, φ' is invariant w.r.t. permutations of $c_{i+1}, \ldots, c_n$.

Proof: Let us first prove the theorem for $i = 0$.

Assume that $\varphi \wedge t = c_0$ is $\mathcal{T}$-satisfiable, and that $\mathcal{M} \in \mathcal{T}$ is a model of $\varphi \wedge t = c_0$; $\mathcal{M}$ is also a model of φ, and thus φ is $\mathcal{T}$-satisfiable.

Assume now that φ is $\mathcal{T}$-satisfiable, and that $\mathcal{M} \in \mathcal{T}$ is a model of φ. By assumption there exists some $j \in \{0, \ldots, n\}$ such that $\mathcal{M} \models t = c_j$, hence $\mathcal{M} \models \varphi \wedge t = c_j$. In the case where $j = 0$, $\mathcal{M}$ is also a model of $\varphi \wedge t = c_0$. If $j \neq 0$, consider the permutation operator P that swaps c_0 and c_j. Notice (this can be proved by structural induction on formulas) that, for any formula ψ, $\mathcal{M} \models \psi$ if and only if $\mathcal{M}_{c_0/d_j, c_j/d_0} \models P[\psi]$, where d_0 and d_j are respectively $\mathcal{M}[c_0]$ and $\mathcal{M}[c_j]$. Choosing $\psi =_{\text{def}} \varphi \wedge t = c_j$, it follows that $\mathcal{M}_{c_0/d_j, c_j/d_0} \models P[\varphi \wedge t = c_j]$, and thus $\mathcal{M}_{c_0/d_j, c_j/d_0} \models P[\varphi] \wedge t = c_0$ since t is invariant w.r.t.

permutations of $c_0, \ldots, c_n$. Furthermore, since φ is invariant w.r.t. permutations of $c_0, \ldots, c_n$, $R[P[\varphi]]$ is φ for the fixed $\mathcal{T}$-preserving rewriting operator. Since R is $\mathcal{T}$-preserving, $\mathcal{M}_{c_0/d_j, c_j/d_0} \models P[\varphi]$ if and only if $\mathcal{M}_{c_0/d_j, c_j/d_0} \models R[P[\varphi]]$, that is, if and only if $\mathcal{M}_{c_0/d_j, c_j/d_0} \models \varphi$. Finally $\mathcal{M}_{c_0/d_j, c_j/d_0} \models \varphi \wedge t = c_0$, and $\mathcal{M}_{c_0/d_j, c_j/d_0}$ belongs to $\mathcal{T}$ since c_0 and c_j are uninterpreted. The formula $\varphi \wedge t = c_0$ is thus $\mathcal{T}$-satisfiable.

For the general case, notice that $\varphi'' =_{\text{def}} \varphi \wedge \neg(t = c_0 \vee \ldots \vee t = c_{i-1})$ is invariant w.r.t. permutations of $c_i, \ldots, c_n$, and $\varphi'' \models_{\mathcal{T}} t = c_i \vee \ldots \vee t = c_n$. By the previous case (applied to the set of constants $c_i, \ldots, c_n$ instead of $c_0, \ldots, c_n$), φ'' is $\mathcal{T}$-equisatisfiable to $\varphi \wedge \neg(t = c_0 \vee \ldots \vee t = c_{i-1}) \wedge t = c_i$. Formulas φ and

$$\big(\varphi \wedge \neg(t = c_0 \vee \ldots \vee t = c_{i-1})\big) \vee \big(\varphi \wedge (t = c_0 \vee \ldots \vee t = c_{i-1})\big)$$

are $\mathcal{T}$-logically equivalent. Since $A \vee B$ and $A' \vee B$ are $\mathcal{T}$-equisatisfiable whenever A and A' are $\mathcal{T}$-equisatisfiable, φ is $\mathcal{T}$-equisatisfiable to

$$\big(\varphi \wedge \neg(t = c_0 \vee \ldots \vee t = c_{i-1}) \wedge t = c_i\big) \vee \big(\varphi \wedge (t = c_0 \vee \ldots \vee t = c_{i-1})\big).$$

This last formula is $\mathcal{T}$-logically equivalent to

$$\varphi \wedge (t = c_0 \vee \ldots \vee t = c_{i-1} \vee t = c_i)$$

and thus the theorem holds. □

Checking if a permutation is syntactically equal to the original term or formula can be done in linear time. And checking if a formula is invariant w.r.t. permutations of given constants is also linear: only two permutations have to be considered instead of the $n!$ possible permutations.

Lemma 6. *A formula φ is invariant w.r.t. permutations of constants $c_0, \ldots, c_n$ if both permutation operators*

- P_{circ} *such that* $P_{\text{circ}}[c_i] = c_{i-1}$ *for* $i \in \{1, \ldots, n\}$ *and* $P_{\text{circ}}[c_0] = c_n$,
- P_{swap} *such that* $P_{\text{swap}}[c_0] = c_1$ *and* $P_{\text{swap}}[c_1] = c_0$

are symmetry operators for φ.

Proof: First notice that any permutation operator on $c_0, \ldots, c_n$ can be written as a product of P_{circ} and P_{swap}, because the group of permutations of $c_0, \ldots, c_n$ is generated by the circular permutation and the swapping of c_0 and c_1. Any permutation P of $c_0, \ldots, c_n$ can then be rewritten as a product $P_1 \circ \cdots \circ P_m$, where $P_i \in \{P_{\text{circ}}, P_{\text{swap}}\}$ for $i \in \{1, \ldots, m\}$. It remains to prove that any permutation operator $P_1 \circ \cdots \circ P_m$ is indeed a symmetry operator. This is done inductively. For $m = 1$ this is trivially true. For the inductive case, assume $P_1 \circ \cdots \circ P_{m-1}$ is a symmetry operator of φ, then

$$\begin{aligned} R[(P_1 \circ \ldots \circ P_m)[\varphi]] &\equiv R[P_m[(P_1 \circ \cdots \circ P_{m-1})[\varphi]]] \\ &\equiv R[P_m[R[(P_1 \circ \cdots \circ P_{m-1})[\varphi]]]] \\ &\equiv R[P_m[\varphi]] \\ &\equiv R[\varphi] \end{aligned}$$

```
1  P := guess_permutations(φ);
2  foreach {c_0, ..., c_n} ∈ P do
3  |  if invariant_by_permutations(φ, {c_0, ..., c_n}) then
4  |  |  T := select_terms(φ, {c_0, ..., c_n}) ;
5  |  |  cts := ∅ ;
6  |  |  while T ≠ ∅ ∧ |cts| ≤ n do
7  |  |  |  t := select_most_promising_term(T, φ) ;
8  |  |  |  T := T \ {t} ;
9  |  |  |  cts := cts ∪ used_in(t, {c_0, ..., c_n}) ;
10 |  |  |  let c ∈ {c_0, ..., c_n} \ cts;
11 |  |  |  cts := cts ∪ {c};
12 |  |  |  if cts ≠ {c_0, ..., c_n} then
13 |  |  |  |  φ := φ ∧ ( ⋁_{c_i ∈ cts} t = c_i );
14 |  |  |  end
15 |  |  end
16 |  end
17 end
18 return φ;
```

Algorithm 1: A symmetry breaking preprocessor.

where $\equiv$ stands for syntactical equality. The first equality simply expands the definition of the composition operator $\circ$, the second comes from the definition of the $\mathcal{T}$-preserving rewriting operator R, the third uses the inductive hypothesis, and the last uses the fact that P_m is either P_{circ} or P_{swap}, that is, also a symmetry operator of φ. □

4 SMT with symmetries: an algorithm

Algorithm 1 applies Theorem 5 in order to exhaustively add symmetry breaking assumptions on formulas. First, a set of sets of constants is guessed (line 1) from the formula φ by the function *guess_permutations*; each such set of constants $\{c_0, \ldots, c_n\}$ will be successively considered (line 2), and invariance of φ w.r.t. permutations of $\{c_0, \ldots, c_n\}$ will be checked (line 3). Notice that function *guess_permutations*(φ) gives an approximate solution to the problem of partitioning constants of φ into classes $\{c_0, \ldots, c_n\}$ of constants such that φ is invariant by permutations. If the $\mathcal{T}$-preserving rewriting operator R is given, then this is a decidable problem. However we have a feeling that, while the problem is still polynomial (it suffices to check all permutations with pairs of constants), only providing an approximate solution is tractable. Function *guess_permutations* should be such that a small number of tentative sets are returned. Every tentative set will be checked in function *invariant_by_permutations* (line 3); with appropriate data structures the test is linear with respect to the size of φ (as a corollary of Lemma 6).

As a concrete implementation of function *guess_permutations*(φ), partitioning the constants in classes that all give the same values to some functions $f(\varphi, c)$ works well in practice, where the functions f compute syntactic information that is unaffected by permutations, i.e. $f(\varphi, c)$ and $f(P[\varphi], P[c])$ should yield the same results. Obvious examples of such functions are the number of appearances of c in φ, or the maximal depth of c within an atom of φ, etc. The classes of constants could also take into account the fact that, if φ is a large conjunction, with $c_0 \neq c_1$ as a conjunct (c_0 and c_1 in the same class), then it should have $c_i \neq c_j$ or $c_j \neq c_i$ as a conjunct for every pair of different constants c_i, c_j contained in the class of c_0 and c_1. In veriT we use a straightforward detection of clusters $c_0, \ldots, c_n$ of constants such that there exists an inequality $c_i \neq c_j$ for every $i \neq j$ as a conjunct in the original formula φ.

Line 3 checks the invariance of formula φ by permutation of $c_0, \ldots, c_n$. In veriT, function *invariant_by_permutations*($\varphi, \{c_0, \ldots, c_n\}$) simply builds, in linear time, the result of applying a circular permutation of $c_0, \ldots, c_n$ to φ, and the result of applying a permutation swapping two constants (for instance c_0 and c_1). Both obtained formulas, as well as the original one, are normalized by a rewriting operator sorting arguments of conjunctions, disjunctions, and equality according to an arbitrary term ordering. The three formulas should be syntactically equal (this is tested in constant time thanks to the maximal sharing of terms in veriT) for *invariant_by_permutations*($\varphi, \{c_0, \ldots, c_n\}$) to return true.

Lines 4 to 15 concentrate on breaking the symmetry of $\{c_0, \ldots, c_n\}$. First a set of terms

$$T \subseteq \{t \mid \varphi \models t = c_0 \vee \ldots \vee t = c_n\}$$

is computed. Again, function *select_terms*($\varphi, \{c_0, \ldots, c_n\}$) returns an approximate solution to the problem of getting all terms t such that $t = c_0 \vee \ldots \vee t = c_n$; an omission in $\mathcal{T}$ would simply restrict the choices for a good candidate on line 7, but would not jeopardize soundness. Again, this is implemented in a straightforward way in veriT.

The loop on lines 6 to 15 introduces a symmetry breaking assumption on every iteration (except perhaps on the last iteration, where a subsumed assumption would be omitted). A candidate symmetry-breaking term $t \in T$ is chosen by the call *select_most_promising_term*(T, φ). The efficiency of the SMT solver is very sensitive to this selection function. If the term t is not important for unsatisfiability, then the assumption would simply be useless. In veriT, the selected term is the most frequent constant-free term (i.e. the one with the highest number of clauses in which it appears), or, if no constant-free terms remains, the one with the largest ratio of the number of clauses in which the term appears over the number of constants that will be required to add to *cts* on line 11; so actually, *select_most_promising_term* also depends on the set *cts*.

Function *used_in*($t, \{c_0, \ldots, c_n\}$) returns the set of constants in term t. If the term contains constants in $\{c_0, \ldots, c_n\} \setminus cts$, then only the remaining constants can be used. On line 10, one of the remaining constants c is chosen nondeterministically: in principle, any of these constants is suitable, but the choice

may take into account accidental features that influence the decision heuristics of the SMT solver, such as term orderings.

Finally, if the symmetry breaking assumption $\bigvee_{c_i \in cts} t = c_i$ is not subsumed (i.e. if $cts \neq \{c_0, \ldots, c_n\}$), then it is conjoined to the original formula.

Theorem 7. *The formula φ obtained after running Algorithm 1 is $\mathcal{T}$-satisfiable if and only if the original formula φ_0 is $\mathcal{T}$-satisfiable.*

Proof: If the obtained φ is $\mathcal{T}$-satisfiable then φ_0 is $\mathcal{T}$-satisfiable since φ is a conjunction of φ_0 and other formulas (the symmetry breaking assumptions).

Assume that φ_0 is $\mathcal{T}$-satisfiable, then φ is $\mathcal{T}$-satisfiable, as a direct consequence of Theorem 5. In more details, in lines 6 to 15, φ is always invariant by permutation of constants $\{c_0, \ldots, c_n\} \setminus cts$, and more strongly, on line 13, φ is invariant by permutations of constants in cts as defined in line 9. In lines 4 to 15 any term $t \in T$ is such that $\varphi \models_{\mathcal{T}} t = c_0 \vee \ldots \vee t = c_n$. On lines 10 to 14, t is invariant with respect to permutations of constants in cts as defined in line 9. The symmetry breaking assumption conjoined to φ in line 13 is, up to the renaming of constants, the symmetry breaking assumption of Theorem 5 and all conditions of applicability of this theorem are fulfilled. □

5 SMT with symmetries: an example

A classical problem with symmetries is the pigeonhole problem. Most SMT or SAT solvers require exponential time to solve this problem; these solvers are strongly linked with the resolution calculus, and an exponential lower bound for the length of resolution proofs of the pigeon-hole principle was proved in [10]. Polynomial-length proofs are possible in stronger proof systems, as shown by Buss [6] for Frege proof systems. An extensive survey on the proof complexity of pigeonhole principles can be found in [13]. Polynomial-length proofs are also possible if the resolution calculus is extended with symmetry rules (as in [12] and in [17]).

We here recast the pigeonhole problem in the SMT language and show that the preprocessing introduced previously transforms the series of problems solved in exponential time with standard SMT solvers into a series of problems solved in polynomial time. This toy problem states that it is impossible to place $n+1$ pigeons in n holes. We introduce n uninterpreted constants $h_1, \ldots, h_n$ for the n holes, and $n+1$ uninterpreted constants $p_1, \ldots, p_{n+1}$ for the $n+1$ pigeons. Each pigeon is required to occupy one hole:

$$p_i = h_1 \vee \ldots \vee p_i = h_n$$

It is also required that distinct pigeons occupy different holes, and this is expressed by the clauses $p_i \neq p_j$ for $1 \leq i < j \leq n+1$. One can also assume that the holes are distinct, i.e., $h_i \neq h_j$ for $1 \leq i < j \leq n$, although this is not needed for the problem to be unsatisfiable.

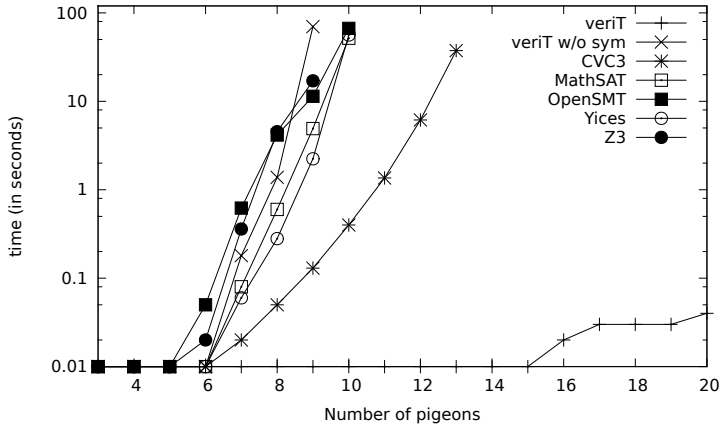

Fig. 1. Some SMT solvers and the pigeonhole problem

The generated set of formulas is invariant by permutations of the constants $p_1, \ldots, p_{n+1}$, and also by permutations of constants $h_1, \ldots, h_n$; very basic heuristics would easily guess this invariance. However, it is not obvious from the presentation of the problem that $h_i = p_1 \vee \ldots \vee h_i = p_{n+1}$ for $i \in [1..n]$, so any standard function *select_terms* in the previous algorithm will fail to return any selectable term to break the symmetry; this symmetry of $p_1, \ldots, p_{n+1}$ is not directly usable. It is however most direct to notice that $p_i = h_1 \vee \ldots \vee p_i = h_n$; *select_terms* in the previous algorithm would return the set of $\{p_1, \ldots, p_{n+1}\}$. The set of symmetry breaking clauses could be

$$
\begin{aligned}
& p_1 = h_1 \\
& p_2 = h_1 \vee p_2 = h_2 \\
& p_3 = h_1 \vee p_3 = h_2 \vee p_3 = h_3 \\
& \quad \vdots \\
& p_{n-1} = h_1 \vee \ldots \vee p_{n-1} = h_{n-1}
\end{aligned}
$$

or any similar set of clauses obtained from these with by applying a permutation operator on $p_1, \ldots, p_{n+1}$ and a permutation operator on $h_1, \ldots, h_n$. Without need for any advanced theory propagation techniques[4], $(n+1) \times n/2$ conflict clauses of the form $p_i \neq h_i \vee p_j \neq h_i \vee p_j \neq p_i$ with $i < j$ suffice to transform the problem into a purely propositional problem. With the symmetry breaking clauses, the underlying SAT solver then concludes (in polynomial time) the unsatisfiability of the problem using only Boolean Constraint Propagation.

Without the symmetry breaking clauses, the SAT solver will have to investigate all $n!$ assignments of n pigeons in n holes, and conclude for each of those assignments that the pigeon $n+1$ cannot find any unoccupied hole.

[4] Theory propagation in veriT is quite basic: only equalities deduced from congruence closure are propagated. $p_i \neq h_i$ would never be propagated from $p_j = h_i$ and $p_i \neq p_j$.

The experimental results, shown in Figure 1, support this analysis: all solvers (including veriT without symmetry heuristics) time out[5] on problems of relatively small size, although CVC3 performs significantly better than the other solvers. Using the symmetry heuristics allows veriT to solve much larger problems in insignificant times. In fact, the modified version of veriT solves every instance of the problem with as many as 30 pigeons in less than 0.15 seconds.

6 Experimental results

In the previous section we showed that detecting and breaking symmetries can sometimes decrease the solving time from exponential to polynomial. We now investigate its use on more realistic problems by evaluating its impact on SMT-LIB benchmarks.

Consider a problem on a finite domain of a given cardinality n, with a set of arbitrarily quantified formulas specifying the properties for the elements of this domain. A trivial way to encode this problem into quantifier-free first-order logic, is to introduce n constants $\{c_1, \ldots, c_n\}$, add constraints $c_i \neq c_j$ for $1 \leq i < j \leq n$, Skolemize the axioms and recursively replace in the Skolemized formulas the remaining quantifiers $Qx.\varphi(x)$ by conjunctions (if Q is $\forall$) or disjunctions (if Q is $\exists$) of all formulas $\varphi(c_i)$ (with $1 \leq i \leq n$). All terms should also be such that $t = c_1 \vee \ldots \vee t = c_n$. The set of formulas obtained in this way is naturally invariant w.r.t. permutations of $c_1, \ldots, c_n$. So the problem in its most natural encoding contains symmetries that should be exploited in order to decrease the size of the search space. The QF_UF category of the SMT library of benchmarks actually contains many problems of this kind.

Figure 2 presents a scatter plot of the running time of veriT on each formula in the QF_UF category. The x axis gives the running times of veriT without the symmetry breaking technique presented in this paper, whereas the times reported on the y axis are the running times of full veriT. It clearly shows a global improvement; this improvement is even more striking when one restricts the comparison to unsatisfiable instances (see Figure 3); no significant trend is observable on satisfiable instances only. We understand this behavior as follows: for some (not all) satisfiable instances, adding the symmetry breaking clauses "randomly" influences the decision heuristics of the SAT solver in such a way that it sometimes takes more time to reach a satisfiable assignment; in any way, if there is a satisfiable assignment, then all permutations of the uninterpreted constants (i.e. the ones for which the formula is invariant) are also satisfiable assignments, and there is no advantage in trying one rather than an other. For unsatisfiable instances, if terms breaking the invariance play a role in the unsatisfiability of the problem, then adding the symmetry breaking clauses always reduces the number of cases to consider, potentially by a factor of $n^n/n!$ (where n is the number of constants), and have a negligible impact if the symmetry breaking terms play no role in the unsatisfiability.

[5] The timeout was set to 120 seconds, using Linux 64 bits on Intel(R) Xeon(R) CPU E5520 at 2.27GHz, with 24 GBytes of memory.

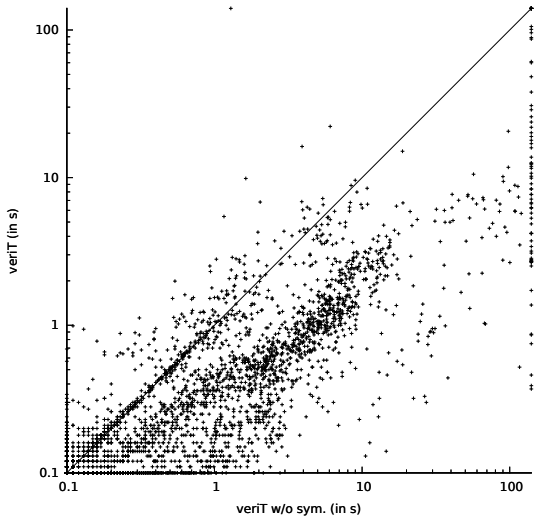

Fig. 2. Efficiency in solving individual instances: veriT vs. veriT without symmetries on all formulas in the QF_UF category. Each point represents a benchmark, and its horizontal and vertical coordinates represent the time necessary to solve it (in seconds). Points on the rightmost and topmost edges represent a timeout.

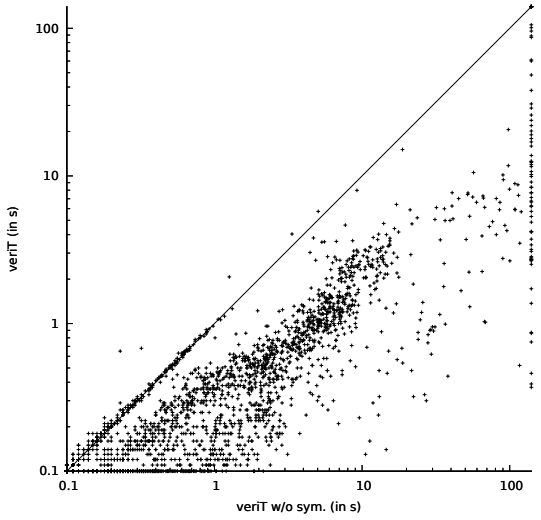

Fig. 3. Efficiency in solving individual instances: veriT vs. veriT without symmetries on the unsatisfiable instances of the QF_UF category.

	Nb. of instances		Instances within time range (in s)						Total time	
	success	timeout	0-20	20-40	40-60	60-80	80-100	100-120	T	T'
veriT	6633	14	6616	9	2	1	3	2	3447	5127
veriT w/o sym.	6570	77	6493	33	14	9	12	9	10148	19388
CVC3	6385	262	6337	20	12	7	5	4	8118	29598
MathSAT	6547	100	6476	49	12	6	3	1	5131	7531
openSMT	6624	23	6559	43	13	6	1	2	5345	8105
Yices	6629	18	6565	32	23	5	1	3	4059	6219
Z3	6621	26	6542	33	23	15	4	4	6847	9967

Table 1. Some SMT solvers on the QF_UF category

To compare with the state-of-the-art solvers, we selected all competing solvers in SMT-COMP 2010, adding also Z3 (for which we took the most recent version running on Linux we could find, namely version 2.8), and Yices (which was competing as the 2009 winner). The results are presented in Table 1. Columns T and T' are the total time, in seconds, on the QF_UF library, excluding and including timeouts, respectively. It is important to notice that these results include the whole QF_UF library of benchmarks, that is, with the diamond benchmarks. These benchmarks require some preprocessing heuristic [16] which does not seem to be implemented in CVC3 and MathSAT. This accounts for 83 timeouts in CVC3 and 80 in MathSAT. According to this table, with a 120 seconds timeout, the best solvers on QF_UF without the diamond benchmarks are (in decreasing order) veriT with symmetries, Yices, MathSAT, openSMT, CVC3. Exploiting symmetries allowed veriT to jump from the second last to the first place of this ranking. Within 20 seconds, it now solves over 50 benchmarks more than the next-best solver.

Figure 4 presents another view of the same experiment; it clearly shows that veriT is always better (in the number of solved instances within a given timeout) than any other solver except Yices, but it even starts to be more successful that Yices when the timeout is larger than 3 seconds. Scatter plots of veriT against the solvers mentioned above give another comparative view; they are available in Appendix A of the full version of this paper [8].

Table 2 presents a summary of the symmetries found in the QF_UF benchmark category. Among 6647 problems, 3310 contain symmetries tackled by our method. For 2698 problems, the symmetry involves 5 constants; for most of them, 3 symmetry breaking clauses were added.

The technique presented in this paper is a preprocessing technique, and, as such, it is applicable to the other solvers mentioned here. We conducted an experiment on the QF_UF benchmarks augmented with the symmetry breaking clauses. We observed the same kind of impressive improvement for all solvers. The most efficient solvers solve all but very few instances (diamond benchmarks excluded): within a time limit of 120s and on the whole library, Yices only fails for one formula, CVC for 36, and the others fails for 3 or 4 formulas. We also observe a significant decrease in cumulative times, the most impressive being

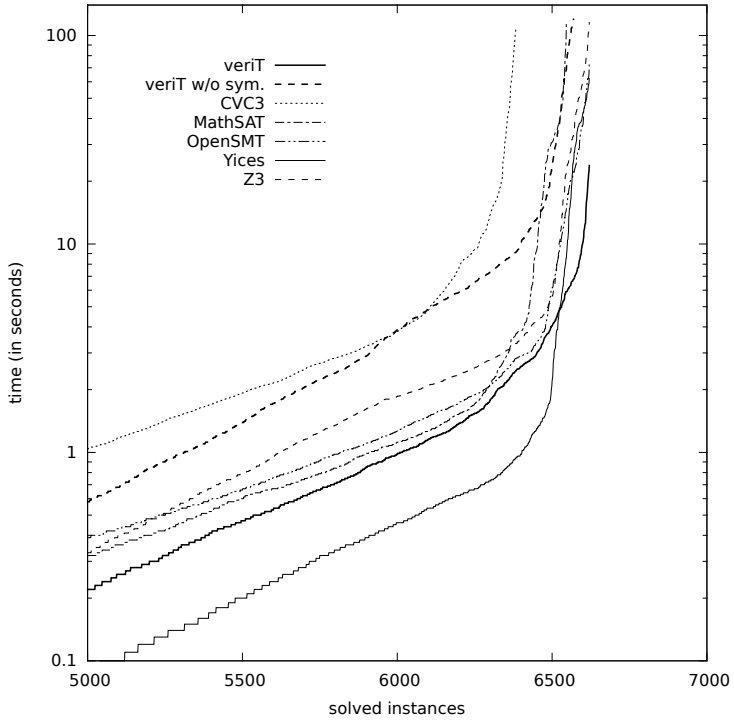

Fig. 4. Number of solved instances of QF_UF within a time limit, for some SMT solvers.

n_c \ n_{sym}	2	3	4	5	6	7	8	9	10	11
1	2									
2		12		8						
3			24	2668						
4				22	92	3				
5					122	166				
6						156				
7							17			
8								11		
9									5	
10										2
Total	2	12	24	2698	214	325	17	11	5	2

Table 2. Symmetries detected for the QF_UF category: n_{sym} indicates the number of constants involved in the symmetry, n_c the number of symmetry breaking clauses.

Yices solving the full QF_UF library but one formula in around 10 minutes. Scatter plots exhibiting the improvements are available in Appendix B of the full version of this paper.

7 Conclusion

Symmetry breaking techniques have been used very successfully in the areas of constraint programming and SAT solving. We here present a study of symmetry breaking in SMT. It has been shown that the technique can account for an exponential decrease of running times on some series of crafted benchmarks, and that it significantly improves performances in practice, on the QF_UF category of the SMT library, a category for which the same solver performed fastest in 2009 and 2010. It may be argued that the heuristic has only be shown to be effective on the pigeonhole problem and competition benchmarks in the QF_UF category. However, we believe that in their most natural encoding many concrete problems contain symmetries; provers in general and SMT solvers in particular should be aware of those symmetries to avoid unnecessary exponential blowup. We are particularly interested in proof obligations stemming from verification of distributed systems; in this context many processes may be symmetric, and this should translate to symmetries in the corresponding proof obligations.

Although the technique is applicable in the presence of quantifiers and interpreted symbols, it appears that symmetries in the other SMT categories are somewhat less trivial, and so, require more clever heuristics for guessing invariance, as well as more sophisticated symmetry breaking tools. This is left for future work. Also, our technique is inherently non-incremental, that is, symmetry breaking assumptions should be retrieved, and checked against new assertions when the SMT solver interacts in an incremental manner with the user. This is not a major issue, but it certainly requires a finer treatment within the SMT solver than simple preprocessing.

The veriT solver is open sourced under the BSD license and is available on `http://www.veriT-solver.org`.

Acknowledgements. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several universities as well as other funding bodies (see `https://www.grid5000.fr`). We would like to thank the anonymous reviewers for their helpful comments and suggestions.

References

1. G. Audemard, A. Cimatti, A. Kornilowicz, and R. Sebastiani. Bounded model checking for timed systems. In D. Peled and M. Y. Vardi, editors, *In IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE)*, volume 2529 of *LNCS*, pages 243–259. Springer, 2002.

2. C. Barrett, L. de Moura, and A. Stump. SMT-COMP: Satisfiability Modulo Theories Competition. In K. Etessami and S. K. Rajamani, editors, *Computer Aided Verification (CAV)*, volume 3576 of *LNCS*, pages 20–23. Springer, 2005.
3. C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS Press, Feb. 2009.
4. P. Baumgartner, A. Fuchs, H. de Nivelle, and C. Tinelli. Computing finite models by reduction to function-free clause logic. *J. Applied Logic*, 7(1):58–74, 2009.
5. T. Bouton, D. C. B. de Oliveira, D. Déharbe, and P. Fontaine. veriT: an open, trustable and efficient SMT-solver. In R. A. Schmidt, editor, *Proc. Conference on Automated Deduction (CADE)*, volume 5663 of *Lecture Notes in Computer Science*, pages 151–156. Springer, 2009.
6. S. R. Buss. Polynomial size proofs of the propositional pigeonhole principle. *Journal of Symbolic Logic*, 52:916–927, 1987.
7. K. Claessen and N. Sörensson. New techniques that improve MACE-style finite model finding. In *Proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications*, 2003.
8. D. Déharbe, P. Fontaine, S. Merz, and B. W. Paleo. Exploiting symmetry in SMT problems, 2011. Available at `http://www.loria.fr/~fontaine/Deharbe6b.pdf`.
9. I. P. Gent, K. E. Petrie, and J.-F. Puget. *The Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, chapter Symmetry in Constraint Programming, pages 329–376. Elsevier, 2006. Edited by Francesca Rossi, Peter van Beek and Toby Walsh.
10. A. Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297 – 308, 1985.
11. X. Jia and J. Zhang. A powerful technique to eliminate isomorphism in finite model search. In U. Furbach and N. Shankar, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, volume 4130 of *LNCS*, pages 318–331. Springer Berlin / Heidelberg, 2006.
12. B. Krishnamurthy. Short proofs for tricky formulas. *Acta Inf.*, 22:253–275, August 1985.
13. A. A. Razborov. Proof complexity of pigeonhole principles. In *Conference on Developments in Language Theory (DLT)*, pages 100–116. Springer-Verlag, 2002.
14. K. Roe. The heuristic theorem prover: Yet another smt-modulo theorem prover. In T. Ball and R. B. Jones, editors, *Computer Aided Verification (CAV)*, volume 4144 of *LNCS*, pages 467–470. Springer, 2006.
15. K. A. Sakallah. Symmetry and satisfiability. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 289–338. IOS Press, Feb. 2009.
16. O. Strichman, S. A. Seshia, and R. E. Bryant. Deciding separation formulas with SAT. In E. Brinksma and K. G. Larsen, editors, *Computer Aided Verification (CAV)*, volume 2404 of *LNCS*, pages 265–279. Springer, July 2002.
17. S. Szeider. The complexity of resolution with generalized symmetry rules. *Theory Comput. Syst.*, 38(2):171–188, 2005.