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 present 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 taking advantage of such symmetries is presented. The implementation of this algorithm in the SMT-solver veriT is used to present the practical benefits of this approach. It results in a significant amelioration of veriT's performances on the SMT-LIB benchmarks that place 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 have been recognized for the satisfiability problem on propositional logic [**?**] and in the area of constraint programming [**?**], to our knowledge, SMT solvers (see [**?**] for a detailed accounting of techniques used in SMT solvers) do not yet fully exploit symmetries. Symmetries in formulas naturally arise while modeling problems that essentially contain symmetries. In the context of SMT solving, a frequent cause for symmetries to appear is when some terms take their value in a finite, given set of totally symmetric elements.

The idea here is very simple: given a formula G left unchanged by all permutations of some uninterpreted constants $c_0, \ldots c_n$, for any model $\mathcal{M}$ of G, if t does not contain these constants and $\mathcal{M}$ makes $t = c_i$ true, there should be a model that set t equal to 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 effective, especially on formulas generated by finite instantiations of quantified problems. As an example, it allows to transform a moderately efficient SMT solver (veriT [**?**]) into a state of the art solver, placing it ahead of the winners of the last editions of the SMT-COMP contest in the QF_UF category. We however do not consider this as a breakthrough in the art of SMT solving, but

rather as an advocacy for ways to specify the symmetries of the problem to automatic provers, just like it is possible to specify symmetries to some constraint programming solvers. Provers may then take the best use of this information to reduce the search space.

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, predicate symbol in $\mathcal{P}$, 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 left 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 $\mathcal{T}$. 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.

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 left unchanged 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* $\mathcal{T} \models G \Leftrightarrow R[G]$ *for any formula* G*. Moreover, for any permutation operator* P*, for any term and any formula,* $R \circ P \circ R$ *and* $P \circ R$ *should yield identical results.*

This last condition will be useful in Lemma 6. Notice that R is 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 capture the notion of $\mathcal{T}$-preserving rewriting operator, assume that the formula contains 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, strictly speaking, 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. *A permutation operator* P *on a language* $\mathcal{L}$ *is a* symmetry operator *of a formula* φ *(a term* t*) on the language* $\mathcal{L}$ *if there exists a* $\mathcal{T}$*-preserving rewriting operator* R *for* P *such that* $R[P[\varphi]]$ *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 satisfying the condition of Def. 3, it is again possible to check in linear time if P is a symmetry operator of a formula.

Symmetries could also have been 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 are new constants). Formulas ψ and $P\psi$ are logically equivalent, that is, P is a semantical symmetry operator of ψ, if and only 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).*

Theorem 5. *Assume given 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 $\varphi \models_{\mathcal{T}} t = c_0 \vee \ldots t = c_n$ then φ is $\mathcal{T}$-satisfiable if and only if*

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

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 formula φ) 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 $\psi =_{\text{def}} \varphi \wedge t = c_j$, $\mathcal{M}_{c_0/d_j, c_j/d_0} \models P[\varphi \wedge t = c_j]$, 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 $c_0, \ldots c_n$, there exists some $\mathcal{T}$-preserving rewriting operator R such that $R[P[\varphi]]$ is φ. 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.

SM: maybe state as separate fact

PF: this seems so trivial I am a bit relunctant to it. What do the other thinks?

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

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

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

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

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

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

and thus the theorem holds. □

Checking if a permutation is syntactically equal to the original 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 with the same $\mathcal{T}$-preserving rewriting operator R.

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 cyclic 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 $P_1 \circ \ldots \circ P_m$, where $P_i \in \{P_{\text{circ}}, P_{\text{swap}}\}$ for $i \in \{1, \ldots, m\}$. It remains to prove that any permutation operator $P_1 \circ \ldots \circ P_m$ is indeed a symmetry operator. This is done inductively. For $m = 1$ this is trivially true. For the other case, assume $P_1 \circ \ldots \circ P_{m-1}$ is a symmetry operator of φ, then

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

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 set of constants is guessed (line 1) from the formula φ by the function *guess_permutations*; each one of those sets $\{c_0, \ldots c_n\}$ of constants will be successively considered (line 2), and invariance of φ w.r.t. permutation of $\{c_0, \ldots c_n\}$ will be checked (line 3). Function

```
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.

guess_permutations(φ) gives an approximate solution of 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, 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*, partitioning the constants in classes that all give the same values to some functions $f(\varphi, c)$ works well in practice: f should then be unaffected by permutations i.e. $f(P\varphi, Pc)$ and $f(\varphi, c)$ should yield the same results. Obvious examples of such functions would be the number of appearances of c in φ, or the maximal depth of c within an atom of φ, ... 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 conjunct (c_0 and c_1 in the same class), it should have $c_i \neq c_j$ or $c_j \neq c_i$ as a conjunct for every different constants c_i, c_j, of the class of c_0 and c_1.

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

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

is computed. Again, function *select_terms*($\varphi, \{c_0, ...c_n\}$) solves an approximation of the problem of getting all terms t such that $t = c_0 \vee \ldots t = c_n$; an omission

in $\mathcal{T}$ would simply restrict the choices for a good candidate on line 7, but would not jeopardize soundness.

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 term $t \in T$ to break symmetry is chosen by the call *select_most_promising_term*(T, φ). This efficiency of the SMT solver is very sensitive to this selection function. If the term t is not important for unsatisfiability, the assumption would simply be useless. In veriT, the term is chosen according to

- the number of appearances in the formula (the higher, the better),
- the number of constants that it will be required to add to *cts* on line 11 (the less, the better); so actually, *select_most_promising_term* also depends on the set *cts*,

with a preference for terms that do not contain any constant in $\{c_0, \ldots c_n\}$.

Function *used_in*$(t, \{c_0, ...c_n\})$ returns the set of constants in term t. If the term contains constants in $\{c_0, \ldots c_n\} \setminus cts$, only the symmetries on the remaining constants can used. On line 10, one of the remaining constant c is chosen non deterministically: this may have a subtle effect on the decision heuristics (for instance, because of arbitrary orderings) in the SMT solver but it is otherwise totally equivalent to take one or another constant.

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 anded to the original formula.

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

Proof: For convenience, the original φ will be denoted φ_0.

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 $\varphi \models_{\mathcal{T}} t = c_0 \vee \ldots t = c_n$. 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 anded 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. Using SMT or SAT solvers to solve this problem will always be exponential; 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 [**?**]. Polynomial-length proofs are possible in stronger proof systems, as shown in [**?**] for Frege proof systems. An extensive survey on the proof complexity of pigeonhole principles can be found in [**?**]. Polynomial-length proofs are also possible if the resolution calculus is extended with symmetry rules (as in [**?**] and in [**?**]).

We here recast the pigeonhole problem to SMT language and show that the previous preprocessing transforms the series of problem solved in exponential time with classical SMT-solvers into a series of problem solved in polynomial time.

This toy problem states that it is impossible to put $n+1$ 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:

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

It is also required that distinct pigeons occupy different holes, and this is expressed by the clauses $p_i \neq p_j$ for $1 \leq i < j \leq n+1$. Without necessity for the unsatisfiability of the problem, one can also assume that the holes are distinct, i.e., $h_i \neq h_j$ for $1 \leq i < j \leq n$.

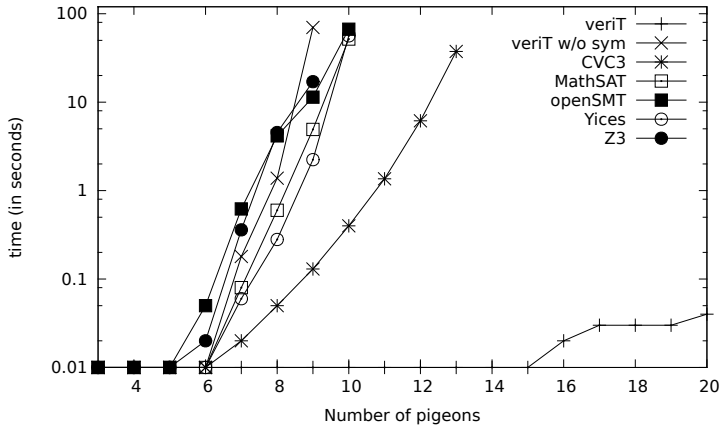

Fig. 1. Some SMT solvers and the pigeonhole problem

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. It is not totally trivial however that $h_i = p_1 \vee \ldots h_i = p_{n+1}$ for $i \in \{1..n\}$, so a non-trivial function *select_terms* in the previous algorithm would 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 $p_i = h_1 \vee \ldots p_i = h_n$; *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

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

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 $h_1, \ldots h_n$. With no 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 underneath SAT solver then concludes (in polynomial time) the unsatisfiability of the problem using only Boolean Constraint Propagation.

Without the symmetry breaking clauses, the underneath SAT solver will 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.

Unsurprisingly, the experimental results match this conclusion. As depicted on Figure 1, all solvers (including veriT without symmetry heuristics) time-out[5] on problems of relatively small size, with CVC3 performing however significantly better. Using the symmetry heuristics allow veriT to solve much larger problems in insignificant times. Not shown on the figure, veriT solves every problem with less than 30 pigeons in less than 0.15 seconds.

6 Experimental results

In the previous section we showed that the technique can decrease the solving time on a series of toy problems from exponential to polynomial. The technique is however not restricted to those toy examples but can indeed improve efficiency on many concrete problems.

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 formula obtained is naturally invariant w.r.t.

[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 $p_i \neq p_j$.

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

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 like these.

Figure 2 presents an scatter plot of the running time of veriT on each formula in the QF_UF category. On the x axis are the running times of veriT without the 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 on unsatisfiable instances (see Figure 3); no significant behavior is observable on satisfiable instances only. We understand this behavior as follow: 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, 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 to try one rather than an other. For unsatisfiable instances, if terms breaking the invariance play a role in the unsatisfiability of the problem, 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.

	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, version 2.8), and Yices (which was competing as the 2009 winner). The results are presented on Table 1. Times T and T' are the total time on the QF_UF library excluding timeouts and including timeouts respectively. It is important to be aware that these results include the whole QF_UF library of benchmarks, that is, with the diamond benchmarks. These benchmarks require some preprocessing heuristic [**?**] which does not seem to be implemented in CVC3 and MathSAT. This accounts for 83 timeouts in CVC3

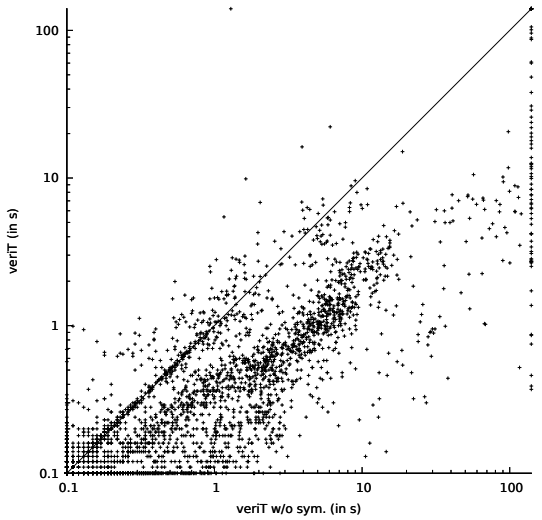

Fig. 2. Efficiency in solving individual instances: veriT vs. veriT without symmetries on the QF_UF category. Each point represents a benchmark, and its horizontal and vertical coordinates represent the time necessary to solve it (in seconds). Points on the rightmost and topmost edges represent a timeout.

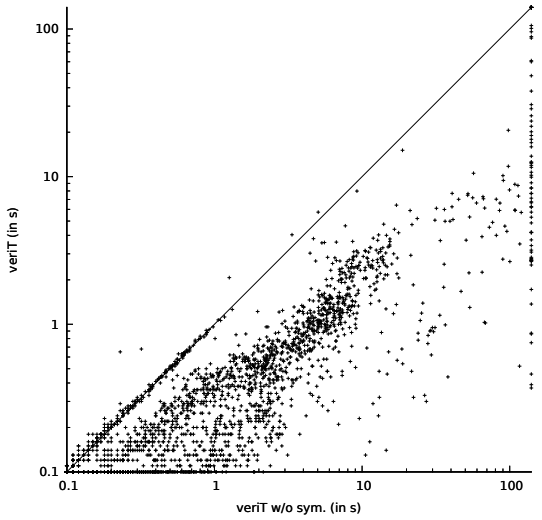

Fig. 3. Efficiency in solving individual instances: veriT vs. veriT without symmetries on the QF_UF category (unsatisfiable instances only).

and 80 in MathSAT. According to this table, with a 120 seconds timeout, the best solvers on QF_UF without the diamond benchmarks are (by decreasing order) veriT with symmetries, Yices, MathSAT, openSMT, CVC3. Exploiting symmetries allowed veriT to jump from the forelast to the first place of this rating. Within 20 seconds, it now solves more than 50 more benchmarks than the second solver.

Figure **??** 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 the other solvers except Yices, but it even starts to be more successful than Yices when the timeout is larger than 3 seconds. The scatter plots on Figure **??** give another comparative view. Again the benefits on the zone with a time smaller that 3 seconds is not always clear. Also, bear in mind that the satisfiable instances do not benefit from the techniques and still exhibit on the scatter plot the somewhat poor efficiency of veriT without symmetries. But the zone between 3 and 120 seconds on the x axis is clearly more populated than the zone between 3 and 120 seconds on the y axis.

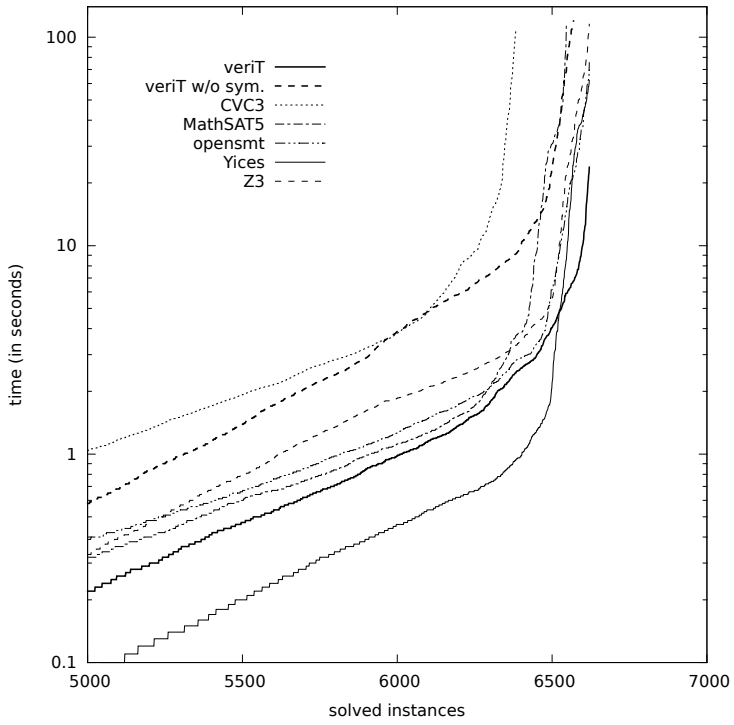

Fig. 4. Number of solved instances of QF_UF within a time limit, for some SMT solvers.

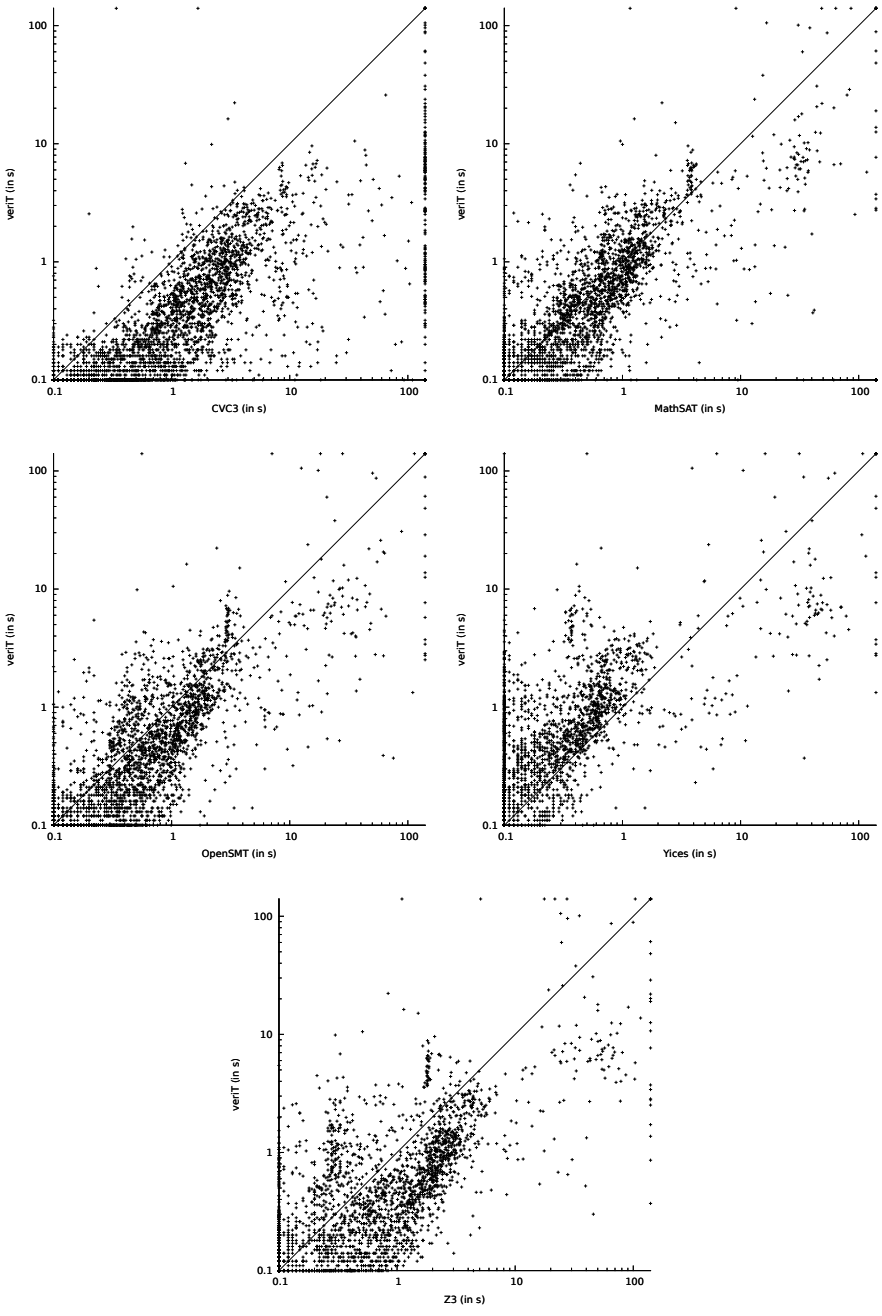

Fig. 5. Efficiency in solving individual instances: veriT vs. some other solvers.

Note to the reviewers: the technique presented in this paper is a preprocessing technique, and, as such, it is applicable to the other solvers mentioned here. It would be informative because it would show if the technique interacts with the other heuristics used in those solvers. However, due to time and computer resources, we were unable to conduct this analysis for the submission deadline. The analysis will be done before the notification for CADE, and will be ready for the camera ready version (if any).

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 showed that the technique can account for an exponential decrease of running times on some series of crafted benchmarks, and showed that it significantly improves performances on the QF_UF category of the SMT library, a category for which last year's winner was also the winner of 2009.

The method presented here could be sarcastically qualified as a heuristic to greatly improve efficiency on the pigeonhole problem and competition benchmarks in the QF_UF category. However we also think that in their most natural encoding many concrete problems do contain many symmetries; provers in general and SMT solvers in particular should be aware of those symmetries to avoid unnecessary exponential blowup.

Although the technique is applicable in presence of quantifiers and interpreted symbols, it seems that symmetries in the other SMT categories are somewhat less trivial, and so, require cleverer invariance guessing heuristics, as well as more sophisticated symmetry breaking tools. This is left for future works. Also, this technique is inherently not incremental, that is, symmetry breaking assumptions should be retrieved, and checked against new assertions when the SMT interacts in an incremental manner. This is not a major issue, but it certainly requires a finer interaction 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.[6]

[6] Note to the reviewers: the current available version is outdated but an updated version will be made available before CADE, with sources.