Fair and Adventurous Enumeration of Quantifier Instantiations

Mikoláš Janota
Czech Technical University in Prague
Prague, Czech Republic
ⓘD

Haniel Barbosa
Universidade Federal de Minas Gerais
Belo Horizonte, Brazil
ⓘD

Pascal Fontaine
University of Liège
Liège, Belgium
ⓘD

Andrew Reynolds
University of Iowa
USA
ⓘD

Abstract—SMT solvers generally tackle quantifiers by instantiating their variables with tuples of terms from the ground part of the formula. Recent enumerative approaches for quantifier instantiation consider tuples of terms in some heuristic order. This paper studies different strategies to order such tuples and their impact on performance. We decouple the ordering problem into two parts. First is the order of the sequence of terms to consider for each quantified variable, and second is the order of the instantiation tuples themselves. While the most and least preferred tuples, i.e. those with all variables assigned to the most or least preferred terms, are clear, the combinations in between allow flexibility in an implementation. We look at principled strategies of complete enumeration, where some strategies are more fair, meaning they treat all the variables the same but some strategies may be more adventurous, meaning that they may venture further down the preference list. We further describe new techniques for discarding irrelevant instantiations which are crucial for the performance of these strategies in practice. These strategies are implemented in the SMT solver cvc5, where they contribute to the diversification of the solver's configuration space, as shown by our experimental results.

Index Terms—SMT, quantifier instantiation, enumeration

I. Introduction

While SMT (satisfiability modulo theory) solvers [5] are used successfully as decision procedures to automatically discharge quantifier-free proof obligations for many applications, there is an increasing need for tools that can furthermore handle quantifiers. Quantified languages however are most often undecidable, or have prohibiting complexity. Quantifier handling within SMT solving is thus a challenge and requires good heuristics.

Quantifier reasoning in SMT builds on the strength of SMT solvers, that is, their ability to efficiently reason on ground formulas, and relies on instantiation: ground consequences of quantified formulas are generated, and the ground reasoner's view of the problem is gradually refined with these instances, to embed knowledge from the quantified formula into ground reasoning. The terms to generate instances may be generated using mostly syntactic methods, e.g., E-matching [6], or semantic techniques like model-based quantifier instantiation [7]. But plain enumeration, done in a principled manner, can give surprisingly good results, particularly in combination with other instantiation techniques [8].

A crucial aspect, when using enumeration-based instantiation, is to prioritize the numerous, often infinite, potential instantiations. When instantiating just one variable, this is essentially a matter of prioritizing smaller terms that are already present in the original formula, according to some order. Quantified assertions however most often have many quantified variables, and there is a lot of freedom on the order on tuples of terms to instantiate those. We here investigate a few strategies based on different tuple orders, some favoring fairness, some being more adventurous, and show that they are valuable in a portfolio of enumerative instantiation strategies. In Section IV, we also present an elimination technique for redundant instantiations that significantly contributes to the improvement of enumeration-based instantiation.

II. Background

Originally, SMT solvers were essentially decision procedures for ground (i.e., quantifier-free) problems in a combination of decidable languages, containing e.g., operators to handle arrays, linear arithmetic expressions, bitvectors, and uninterpreted predicates and functions. They excel at deciding the satisfiability of large formulas in these languages. As a toy example, consider the (satisfiable) conjunctive set of formulas

$$\{R(a), \neg S(b), a = b\}.$$

It belongs to the quantifier-free fragment of first-order logic, and as such, is decided by many SMT solvers. Quantifier reasoning in modern SMT solvers builds on this. The input formula, possibly after a pre-processing phase, is first given to the ground solver. From the point of view of this ground solver, each quantified formula is abstracted into a distinct propositional variable. As an example, the conjunctive set

$$\{R(a), \neg S(b), a = b, \forall x . R(x) \Rightarrow S(x)\}$$

is understood by the ground solver as the previous ground set, augmented with an abstract proposition Q corresponding to $\forall x . R(x) \Rightarrow S(x)$. Then the ground solver provides a satisfying assignment for the ground part of the formula, including a valuation of the propositional variables abstracting the quantified formulas (in our case Q must be true). The instantiation module recovers the quantified formulas associated to these variables, and generates new instances of the quantified formulas to the ground reasoner (Figure 1). In our toy example such an instance could be
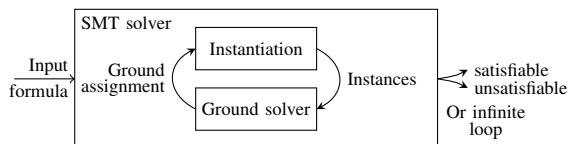
$$Q \Rightarrow (R(a) \Rightarrow S(a)),$$

Fig. 1. The SMT instantiation loop.

which would render the problem unsatisfiable at the ground level. In general, the instantiation loop is iterated until the ground reasoner is able to conclude that the formula is unsatisfiable, a time out is reached, or no instance can be deduced anymore. In this paper, we focus on refutations only and will not consider the last case.

Thanks to the Herbrand Theorem (see e.g., [8]), with fair enumeration of instances using all possible terms built on the appropriate set of symbols, SMT solving is refutationally complete for satisfiability modulo well-behaved first-order theories. Since typical SMT inputs contain hundreds of quantified formulas with many nested quantifiers, on a language with often infinitely many terms, the number of possible instances is very large, and most often infinite. It is crucial to quickly find out the right instances, otherwise the ground solver will be overwhelmed by the amount of instances. For a quantified formula $\forall x_1 \ldots x_n . \varphi$ with n variables, this boils down to ordering n-tuples of ground terms to prioritize instantiation.

III. ENUMERATION STRATEGIES

We start by the assumption that for each variable x_i there is a sequence of terms $\mathcal{T}_i = t_i^1, t_i^2, \ldots$, which are the possible candidates for instantiation into the variable x_i. We further assume that this sequence of terms is sorted by some given preference, i.e., that t_i^j is more likely to yield a useful instantiation than the candidate $t_i^{j'}$ with $j < j'$. This lets us focus on the indices into the sequences of terms, rather than on the terms themselves. An instantiation, i.e., a tuple of terms, is uniquely represented as an n-tuple of indices.

While this setup already assumes a given order on the terms for the individual variables, it does not tell us how to order the actual tuples. Clearly, the tuple of indices $(0, \ldots, 0)$ is the most advantageous and $(|\mathcal{T}_1| - 1, \ldots, |\mathcal{T}_n| - 1)$ is the least advantageous one. However, it is unclear whether $(0, 1, 1)$ is more advantageous than $(0, 0, 2)$, or the other way around. This motivates our quest for different enumeration strategies. A general notion from multi-objective optimization is useful: *Pareto-optimal* solutions are such that improving any criterion worsens some other.
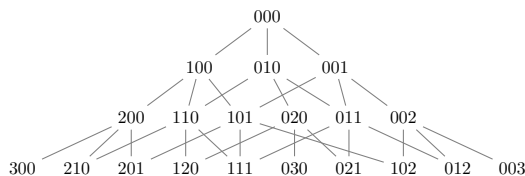

Fig. 2. Pareto graph for 3 variables with 4 candidate terms for each.

Definition 1 (Pareto dominates)**.** *Let* $t_1 = (a_1, \ldots, a_n)$ *and* $t_2 = (b_1, \ldots, b_n)$ *be* n-*tuples of integers. We say that* t_1 *Pareto dominates* t_2, *if and only if* $t_1 \neq t_2$ *and* $a_i \leq b_i$ *for all* $i \in 1..n$.

We focus on traversals of the graph of tuples where traversing an edge increases one of the indices. Hence, there is an edge from tuple t_1 to tuple t_2 iff t_2 is obtained by increasing either of the digits of t_1 by 1; see Figure 2. This graph anchors our initial motivation that the order on the terms pertaining to a single variable represents preference. Indeed, following down any edge in this graph means going to a less preferred tuple. We call this graph the *Pareto graph*.

So what does differentiate one traversal from another? In graph theory vernacular, a traversal is broad or deep. In our context, a broad traversal is more *fair* since it alters terms of different variables evenly. A deep traversal is more *adventurous* since it opts for less preferred, i.e., riskier, instantiations.

Fair strategies observe the Pareto ordering, meaning that no tuple dominates any of the previous tuples. For instance, the sequence $(0, 0), (0, 1), (1, 0), (1, 1)$ respects Pareto ordering but $(0, 0), (0, 1), (1, 1), (1, 0)$ does not because $(1, 0)$ Pareto-dominates $(1, 1)$. Note that both of these examples respect the Pareto graph in the sense that a node is visited only if at least one of its predecessors has been visited.

In the remainder of the section we introduce techniques considered in the experimental evaluation in Section V. On a technical note, in practice the number of possible candidates per variable may vary, but for the sake of clarity, we assume that each variable has the same number of possible candidate terms. This means that every element of the tuple (digit) is in the range $0..M$ for some fixed $M \in \mathbb{N}$. Effectively, this means that we are looking for systematic enumerations of tuples from the space $[0..M]^n$, with a fixed set of n variables.

A. Stages by maximal digit [8]

This ordering interprets tuples as numbers in increasing base $b \in 2..(M + 1)$. As an example, consider two variables and $M = 2$. The enumeration starts with base 2, yielding: $(0, 0), (1, 0), (0, 1), (1, 1)$. Subsequently, it switches to base 3, while skipping already enumerated tuples, giving the rest of the tuples: $(2, 0), (2, 1), (0, 2), (1, 2), (2, 2)$.

This is a natural alternative to interpreting the tuples as numbers in base $M + 1$, which would lead to a highly unfair strategy because large values of M would lead to changing significant digits very late.

This ordering observes Pareto domination and the enumeration algorithm runs in constant space.

B. Stages by sum of digits

The maximum digit approach mitigates unfairness in large value of M (large number of candidate terms). However, it still leads to an imbalance with a large number of quantified variables, i.e., with large tuples. Indeed, even with $M = 1$ already 10 variables require 2^{10} iterations before the most significant digit is changed. The alternative is to iterate over combinations stratified by the *sum of all the digits*. Tuples with the same sum of digits are ordered lexicographically.

This leads to a breadth first traversal of the Pareto graph and its effect is more pronounced with large number of variables. The initial sequence has the following form:

$$(0, 0, \ldots, 0), (1, 0, \ldots, 0), (0, 1, \ldots, 0), \ldots, (0, 0, \ldots, 1),$$
$$(2, 0, \ldots, 0), (1, 1, \ldots, 0), (0, 2, \ldots, 0), \ldots$$

This ordering also observes the Pareto domination and can be calculated in constant space.

C. Leximax

Arguably the most fair strategy is enumeration according to the *leximax order* [1] since all the variables are in equivalent roles: let t_1, t_2 be n-tuples of integers. We say that t_1 is *leximax preferred* to t_2 if $t_1^{\downarrow}$ is lexicographically smaller than $t_2^{\downarrow}$, where $t^{\downarrow}$ denotes t sorted in descending order. Enumeration can be done in constant space. We observe that all permutations of a tuple are incomparable. This enables us to stage the enumeration by gradually worsening a sorted tuple and enumerate all its permutations through standard means. The incomparable permutations are enumerated lexicographically. For two variables the sequence starts as follows, $(0, 0), (0, 1), (1, 0), (1, 1), (0, 2), (2, 0)$. Contrast that with the sum of digits $(0, 0), (0, 1), (1, 0), (0, 2), (1, 1), (2, 0)$.

D. Iterative Deepening and Random-walk Search

Strategies discussed so far never violate Pareto domination, which would be violated by depth-first but that would have a large degree of unfairness. Instead, we propose to use *iterative deepening* where the maximum depth is incremented by some fixed parameter $k \in \mathbb{N}^+$. Maximum depth 2 yields $(0, 0), (0, 1), (0, 2), (1, 1), (1, 0), (2, 0)$, where $(1, 0)$ Pareto-dominates $(1, 1)$, even though it comes later in the sequence.

As another very adventurous strategy, we propose *random-walk traversal*, which is similar to DFS but instead of a stack we use a set where the next element is chosen randomly.

IV. DISCARDING REDUNDANT INSTANTIATIONS

When solving quantified formulas, SMT solvers are often hindered by an overabundance of generated instantiations. Thus, it is paramount to avoid instantiations that are *redundant*. At a high level, an instantiation is considered redundant if it does not help rule out models in the current context. Methods for discovering redundant instantiations are particularly important in the context of enumerative instantiation, where typically we are iterating over similar domains of terms on multiple instantiation rounds, and are looking for the first instantiation that is not redundant.

In our implementation, we consider three criteria for determining that an instantiation $\varphi \cdot \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ is redundant, in increasing order of cost:

1) (Duplicate Term Vector) For each φ, maintain a trie containing all term vectors of its previous instantiations. If $(t_1, \ldots, t_n)$ is already in this trie, then the instantiation is redundant.
2) (Entailed) As described in [8, Section 4.1], a fast incomplete method for entailment is used for discovering when an instantiation lemma is already implied by the current set of constraints known by the SMT solver. All instantiations that are entailed are considered redundant.
3) (Duplicate Formula Modulo Rewriting) Maintain a set of previous formulas returned by quantifier instantiation. Construct the formula $\varphi \cdot \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ and normalize it using rewriting techniques. If the resulting formula is already in our set, it is redundant.

If none of these criteria hold, the instantiation is not considered redundant.

It is important to note that the latter two methods allow one to learn that a *class* of instantiations is redundant. For this purpose, we introduce the concept of a *fail mask* for an instantiation. A fail mask $\mathcal{M}$ for a substitution $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ is a sequence of n bits such that all substitutions that extend $\{x_i \mapsto t_i \mid$ the ith bit of $\mathcal{M}$ is set $\}$ when applied to φ result in a redundant instantiation.

For example, let φ be the formula $P(x_1, x_2) \vee Q(x_2, x_3)$, and consider the substitution $\sigma = \{x_1 \mapsto a, x_2 \mapsto b, x_3 \mapsto c\}$. Let $E = \{P(a, b), \neg Q(b, c)\}$ be the current set of assertions from the ground solver. The instantiation $\varphi \cdot \sigma$ is redundant; a fail mask for σ is 110, since $P(a, b) \vee Q(b, x_3)$ is entailed by E for any value of x_3.

We incorporate fail masks into our implementation in the following way. When an instantiation $\varphi \cdot \sigma$ is discovered to be redundant, we construct the fail mask $\mathcal{M}$ containing all 1s. Starting with $i = 1$, we drop the entry $\{x_i \mapsto t_i\}$ from σ. If the instantiation is still redundant based on the latter two criteria above, then we set the ith bit to 0. If not, then we re-add the entry $\{x_i \mapsto t_i\}$ to σ, and proceed with $i + 1$. Notice this means that our computation of the fail mask is greedy.

The fail mask is incorporated into the enumerative strategies as follows. After each failed instantiation, combine the tuple of term indices and the fail mask into a tuple with wildcards, denoted "?". So for instance, if the tuple $(5, 4, 3)$ fails with the mask 101, construct the tuple $(5, ?, 3)$ meaning that if the first variable is instantiated with the 5th term and the third variable with the 3rd term, the instantiation is bound to be redundant. Such combinations we wish to avoid. This is checked independently of the enumeration algorithm by storing the disabled patterns into a trie and discarding any combinations matching one of the previously disabled patterns. The trie handles the wildcard character ? specially by always matching on it.

V. EXPERIMENTS

This section reports on our experimental evaluation of different tuple enumeration strategies implemented in the cvc5 SMT solver (the successor of CVC4 [3]). We performed all experiments on a cluster with Intel Xeon CPU E5-2620 CPUs with 2.1GHz and 128GB memory, providing one core, 300 seconds, and 8GB RAM for each job.

Enumerative instantiation is extensively compared with other techniques in [8], where it was concluded that interleaving E-matching with enumeration gives the best results. However, as the focus of the paper is the different enumeration

Library	#	e	u	id2	id4	lmax	sum	rwlk	allu-port	eu-port	eallu-port	z3
TPTP	18627	**7765**	6989	6801	6834	6832	6922	6839	7330	9056	9292	-
UF	7668	**3243**	3016	2975	2963	2959	3009	2992	3120	3433	3452	2905
UFLIA	10137	**7424**	6024	6018	5897	6001	5980	5994	6188	7595	7615	6912
UFNIA	13509	5715	**7458**	7396	7384	7426	7437	7430	7620	7740	7843	6491

strategies, we run enumeration on its own. For succinctness, we omit certain details, such as relevant domain heuristic, run as proposed in [8].

Benchmarks are selected from first-order benchmarks from the TPTP library [10], version 7.4.0, and from SMT-LIB [4], 2020 release. Of 19287 first-order TPTP problems, we excluded 660 which contained polymorphic types, leaving 18627 for consideration. For SMT-LIB, we considered all problems from logics containing quantifiers and integer arithmetic, i.e., UF, UFLIA, and UFNIA, totaling 31314 problems. This selection of benchmarks was inspired by the evaluation from [8], where enumerative instantiation was shown more effective in the above sets.
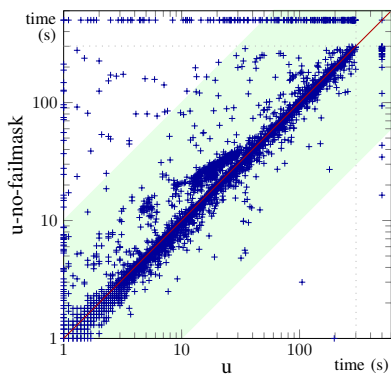

Fig. 3. Impact of elimination of redundant instantiation via fail masks.

The evaluation covers a number of cvc5 configurations. The default enumeration, maximal digit, is denoted as **u**. Its variations according to different enumeration strategies described above are id-n for iterative deepening with increment n; **lmax** for leximax; **sum** of digits; and **rwlk** for random walk. We also run, for control, cvc5's E-matching (denoted **e**) and z3 4.8.10 (denoted **z3**). By default z3 uses a combination of E-matching and model-based quantifier instantiation. All the cvc5 configurations run with the fail-masks technique enabled; further, they use conflict-based instantiation [2], [9] as a "fail-fast" technique, given its strong focusing effect. The implementation of E-matching in cvc5 already uses a redundancy checking mechanism [2], which is always enabled in our experiments. The z3 evaluation is restricted to SMT-LIB, given its limited support for TPTP.

The results are summarized in Table I. The column **allu-port** is a virtual best solver (**vbs**) of all the enumerative configuration, **eu-port** of a vbs of only e and u, and eallu-

Library	#	e	u	id2	id4	lmax	sum	rwlk	z3
TPTP	18627	**1862**	27	5	22	12	14	17	-
UF	7668	**160**	0	0	5	3	1	2	126
UFLIA	10137	**370**	0	3	1	3	1	1	49
UFNIA	13509	76	3	8	9	12	2	9	**547**

port a vbs of all cvc5 configurations. We first emphasize the tremendous advantage in UFNIA of u over e, which can be explained by many benchmarks needing instantiations with key arithmetic constants, such as 0, to enable the necessary ground reasoning to solve the problem. However, a large number of these benchmarks may be impossible to solve via E-matching alone: if matching needs to be done on terms containing arithmetic operators, e.g. to match $x + 1$ with 1, E-matching will fail, whereas enumerative instantiation would instantiate the formula regardless. Moreover, the different enumeration strategies do lead to significant orthogonality among the different configurations. The number of uniquely solved problems per strategy is shown in Figure II. Note also that the vbs of the enumerative configurations versus u reduces the number of *unsolved* problems in UFNIA in almost 3%, while eallu-port vs eu-port reduces the number of unsolved in almost 2%. These improvements are also present in TPTP, with similar reductions in the number of unsolved problems when considering all the enumeration strategies in a virtual best solver. This clearly shows the benefit of integrating into actual portfolios different enumeration strategies rather than having just the default one.

We also evaluated an even more adventurous enumeration strategy than those in Table I, which randomly changes the strategy at each instantiation round, thus effectively simultaneously trying all the strategies. This random strategy performs similarly to the others but can be deeply influenced by the random seed chosen for selecting a strategy each round, to the extent that changing the seed from 0 to 7 makes it go, in UFLIA, from 6007 successes to 6047. This further reinforces the usefulness of diversifying the set of strategies used for quantifier instantiation in practice.

Discarding classes of redundant instantiations using fail masks gives a clear advantage as illustrated in Figure 3 (default enumerative instantiation strategy, on all benchmarks). Using the fail masks leads to 217 uniquely solved problems, whereas without it only 31 problems are solved uniquely.

Moreover, a large number of commonly solved problems have very significant speed-ups, as the plot makes clear. These improvements can be explained by the technique being the most effective in problems containing quantifiers with many variables, which are common occurrences among the benchmark sets we considered. On problems where the fail masks do not help, the overhead of computing and checking them is noticeable (see the often prevalent crosses just below the diagonal line). However, it is far from a deterrent, given the significant gains.

VI. Conclusions

Enumerative instantiation is powerful, versatile, and offers a lot of freedom for strategies. We presented several ordering heuristics for instantiation that contribute to the orthogonality of the strategies, and ultimately improve the SMT solver's performance and robustness. This is especially useful when a user is willing to employ a barrage of solver configurations to tackle a high-priority problem instance.

In future work, we plan to investigate the applications of enumerative instantiation strategies for portfolio approaches to SMT solving. We also would like to pursue more advanced techniques where tuple and term orderings are not fixed and may be influenced by previous successes or failures.

Acknowledgments

References

[1] Salvador Barbarà and Matthew Jackson. Maximin, leximin, and the protective criterion: Characterizations and comparisons. *Journal of Economic Theory*, 46(1):34–44, 1988.

[2] Haniel Barbosa, Pascal Fontaine, and Andrew Reynolds. Congruence closure with free variables. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for Construction and Analysis of Systems (TACAS), Part II*, volume 10206 of *Lecture Notes in Computer Science*, pages 214–230, 2017.

[3] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification (CAV)*, pages 171–177. Springer, 2011.

[4] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at www.SMT-LIB.org.

[5] Clark W. Barrett and Cesare Tinelli. Satisfiability modulo theories. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking.*, pages 305–343. Springer, 2018.

[6] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[7] Yeting Ge and Leonardo Mendonça de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *Computer Aided Verification, 21st International Conference, CAV*, pages 306–320, 2009.

[8] Andrew Reynolds, Haniel Barbosa, and Pascal Fontaine. Revisiting enumerative instantiation. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 10806, pages 112–131, 2018.

[9] Andrew Reynolds, Cesare Tinelli, and Leonardo Mendonça de Moura. Finding conflicting instances of quantified formulas in SMT. In *Formal Methods In Computer-Aided Design (FMCAD)*, pages 195–202. IEEE, 2014.

[10] Geoff Sutcliffe. The TPTP problem library and associated infrastructure. *J. Autom. Reasoning*, 43(4):337–362, 2009.