Rewrites for SMT Solvers using Syntax-Guided Enumeration (Work in Progress)*

Andrew Reynolds¹, Haniel Barbosa¹, Aina Niemetz², Andres Nötzli², Mathias Preiner², Clark Barrett², and Cesare Tinelli¹

¹ University of Iowa, Iowa City, USA
andrew.j.reynolds@gmail.com,{haniel-barbosa,cesare-tinelli}@uiowa.edu
² Stanford University, Stanford, USA
{niemetz,noetzli,preiner,barrett}@cs.stanford.edu

Abstract

In this paper, we explore a development paradigm for SMT solver developers where rewrite rules are suggested to the developer using syntax-guided enumeration. We capitalize on the recent advances in enumerative syntax-guided synthesis (SyGuS) techniques for efficiently enumerating terms in a grammar of interest, and novel sampling techniques for testing equivalence between terms. We present our preliminary experience with this feature in the SMT solver cvc4, showing its impact on its rewriting capabilities using several internal metrics, and its subsequent impact on solving bit-vector and string constraints.

1 Introduction

Developing a state-of-the-art SMT solver is a demanding process. In particular, implementing a basic decision procedure for a specific theory is not enough, as in practice many problems can be solved only after they have been simplified to a form for which the SMT solver is effective. We refer to the component of the SMT solver that performs this simplification as its rewriter, which given a term \( t \) returns the rewritten form of that term, denoted by \( t\downarrow \). Designing an effective and correct rewriter is challenging. It requires extensive domain knowledge and an analysis of specific problem instances. New rewrites are often created when a set of constraints cannot be solved unless it is first simplified in a particular way. Depending on the theory, optimizing the implementation of its rewriter can be as important, or even more important, than optimizing its decision procedure. Moreover, recent work [21,20] has shown a generic inference scheme can be integrated into CDCL(\( T \)) theory solvers based on using its rewriter as an oracle.

In this paper, we explore a development paradigm where the SMT solver itself guides the developer in the implementation of the rewriter. Our twofold goal is to improve the performance of SMT solvers on constraints of importance to applications and increase the productivity of the SMT solver developer during this task. This preliminary work focuses mostly on the latter goal. Our approach differs from automatic rewrite rules generators, such as SWAPPER [24] and Hazel [15], by being more flexible and application-independent: the suggestion of rewrite rules is parameterized by a user-defined grammar, which specifies a set of terms to be rewritten, rather than by particular problems.

A Workflow for Developing Rewrite Rules in an SMT Solver An overview of our workflow is shown in Figure 1. In Step 1, the developer provides a grammar and specification to the SMT solver.¹ Conceptually, this input describes the class of terms that the developer is interested in targeting in the rewriter. In Step 2, we use previous techniques (namely the syntax-guided synthesis module of the SMT solver cvc4 [19])

---

*This work was partially supported by DARPA under award No. FA8750-15-C-0113, and the National Science Foundation under Award 1656926.

¹ We give examples of user-provided grammars in Section 5.
to efficiently enumerate target terms into a term database. In the third step, we pair those terms together to form a candidate rewrite database. We then report to the developer, in Step 4, a subset of this database as a set of unoriented pairs \( t_1 \approx s_1, \ldots, t_n \approx s_n \). These pairs have the following two key properties:

1. terms \( t_i \) and \( s_i \) were inferred to be equivalent based on some criteria in Step 3, and
2. \( t_i \downarrow \) is not equal to \( s_i \downarrow \), in other words, the current rewriter does not treat \( t_i \) and \( s_i \) as equivalent.

Hence, this list can be understood as a to do list for the developer of the rewriter. In Step 5 of the workflow, the developer uses this list to extend the implementation on the rewriter so that it rewrites more pairs of equivalent terms to the same term. We stress that our goal is not to automate the implementation of the rewriter. We have found that the workflow is most effective when candidate rewrite rules act as hints to inspire the developer, who, through creativity and careful engineering, is subsequently able to implement a better rewriter in Step 5. Automating this step is left to future work.

Outline We describe the steps of our workflow in more depth in the following. Steps 2, 3 and 4 are described in Sections 2, 3 and 4 respectively. In Section 5, we give an overview of our first experience with this workflow in the cvc4 solver [4], and give an initial evaluation of how improving the rewriter can lead to improvements in SMT solving. We discuss related work in Section 6 and future work in Section 7.

2 Syntax-Guided Synthesis for Term Enumeration

In this section, we describe how syntax-guided techniques are used for enumerating terms in Step 2 in the workflow of Figure 1. For this task, we leverage existing work on highly optimized enumerative approaches to the syntax-guided synthesis problem [19].

A syntax-guided synthesis problem for a function \( f \) in a background theory \( T \) consists of:

1. a set of syntactic restrictions, given by a grammar \( R \), and
2. a set of semantic restrictions, or specification, given by a (second-order) \( T \)-formula of the form \( \exists f. \forall \vec{x}. \varphi[f, \vec{x}] \) where \( \varphi \) is (typically) a quantifier-free formula.

Formally, a grammar \( R \) is a triple \( (s_0, S, R) \) where \( s_0 \) is an initial symbol, \( S \) is a set of symbols with \( s_0 \in S \), and \( R \) is a set of rules \( s \rightarrow t \), where \( s \in S \) and \( t \) is a term built from symbols in \( T \)'s signature, free
variables, and symbols from $S$. The rules define a rewrite relation over such terms, also denoted by $\rightarrow$, as expected. We say a term $t$ is generated by $\mathcal{R}$ if $s_0 \rightarrow^* t$ where $\rightarrow^*$ is the reflexive-transitive closure of $\rightarrow$ and $t$ contain no symbols from $S$. A solution for $f$ is a closed lambda term $\lambda \bar{y}. t$ of the same type as $f$ such that $t$ is generated by $\mathcal{R}$ and $\forall \bar{x}. \varphi[\lambda \bar{y}. t, \bar{x}]$ is valid in $T$ (modulo beta-reductions).

An enumerative approach to syntax-guided synthesis can be divided into two components:

1. a candidate solution generator, which produces a stream of terms $t_1, \ldots, t_n, \ldots$ in the language generated by grammar $\mathcal{R}$, and
2. a verifier which, given a candidate solution $t_i$, checks whether $\forall \bar{x}. \varphi[\lambda \bar{y}. t_i, \bar{x}]$ is valid in $T$.

The synthesis solver terminates as soon as it generates a candidate solution $t_i$ that passes the test in Step 2 above. In practice, most state-of-the-art enumerative syntax-guided synthesis solvers [25, 2] are implemented as a layer on top of an SMT solver with support for quantifier-free $T$-formulas, which can be used as a verifier in this approach. An exception is the syntax-guided synthesis solver of cvc4, where a single instance of the SMT solver acts as both the candidate solution generator and the verifier [19].

The second step in our workflow in Figure 1 is to generate a database of terms that are eligible to appear in rewrite rules. We observe that the existing enumerative syntax-guided synthesis solver of cvc4 can be used as a term generator in this step. In particular, the syntax-guided synthesis solver of cvc4 can be instrumented to produce multiple solutions $t_1, t_2, \ldots$ to a syntax-guided synthesis problem specified in Step 1 of the workflow. We do this by adding a clause that blocks the current solution when one is found. Each of these solutions is added to a term database.

The specification $\exists \bar{f}. \forall \bar{x}. \varphi[\bar{f}, \bar{x}]$ acts as a filtering mechanism, to discard terms that should not be included in rewrite rules. For example, the specification $\exists \bar{f}. \forall x_1 x_2. f(x_1, x_2) \approx f(x_2, x_1)$ indicates that we are only interested in enumerating terms that correspond to commutative functions. In contrast, if we do not wish to impose any semantic restrictions, we can provide $\exists \bar{f}. \top$ as a specification, which amounts to telling the system that we are interested in any terms that meet syntactic restrictions given by $\mathcal{R}$.

Another key property of the syntax-guided synthesis solver of cvc4 is that it only enumerates solutions that are unique according to the current rewriter. The motivation for this, in the context of syntax-guided synthesis, is that if we enumerate a candidate solution $t_1$ and $\forall \bar{x}. \varphi[\lambda \bar{y}. t_1, \bar{x}]$ is not $T$-valid, then it is fruitless to try another solution $t_2$ that is $T$-equivalent to $t_1$ since $\forall \bar{x}. \varphi[\lambda \bar{y}. t_2, \bar{x}]$ is also not $T$-valid. In cvc4’s synthesis solver, the rewriter is used as an oracle to under-approximate term equivalence in the theory $T$ (where the syntactic equality of $t_1 \downarrow$ and $t_2 \downarrow$ implies the equivalence of $t_1$ and $t_2$ in $T$). When the syntax-guided synthesis solver of cvc4 discovers the equivalence of two terms $t_1$ and $t_2$, it adds symmetry breaking constraints to ensure that subsequent solutions do not include $t_1$ (resp., $t_2$) as a subterm (see Section 5.1 of [18] for a recent description of these techniques). As a consequence, cvc4 will never generate two candidate solutions (and hence solutions) that are identical up to rewriting.

This last property is highly convenient in the context of Figure 1. Concretely, it ensures that the term database generated as a result of Step 2 contains no terms $t_1$ and $t_2$ such that $t_1 \downarrow = t_2 \downarrow$. This in turn ensures that the rewrite rules generated in the next step are not redundant, that is, they do not rewrite a term $t_i$ to a term $t_j$ where $t_i$ already has the same rewritten form as $t_j$.

## 3 Equivalence Checking Techniques for Rewrite Rule Generation

In Step 3 of the workflow in Figure 1, we are given a database of terms (possibly containing free variables $\bar{y}$) generated by syntax-guided enumeration in the form of a set $D$, which we call our term database. From this set, we generate a set of (unoriented) pairs of the form $t_1 \approx s_1, \ldots, t_n \approx s_n$ where $\bar{y}$ lists the formal arguments of the function $f$ to be synthesized as specified in Step 1.
We call such pairs candidate rewrite rules. A naïve method for constructing candidate rewrite rules from the set $D$ would be to consider each distinct pair of terms $t_i, t_j$ in $D$, check the satisfiability of $\exists \bar{y}. t_i \neq t_j$, and return $t_i \approx t_j$ if this disequality is unsatisfiable. Doing so is highly inefficient in practice, and may even be infeasible depending on the background theory $T$. For instance, if $T$ is the theory of strings with length, whose decidability is unknown [9], a single check of this form may be non-terminating. Instead, we have developed novel techniques that are based on testing the evaluation of terms $t_i$ and $s_i$ on a set of concrete sample points.

In detail, our implementation of Step 3 of Figure 1 in cvc4 maintains an equivalence relation between the terms in our (evolving) term database $D$. Each equivalence class $\{t_1, \ldots, t_n\}$ in this relation is such that, for each $i = 1, \ldots, n$, a pair of the form $t_i \approx t_j$ has been generated for some $j$. In other words, this equivalence relation corresponds to the transitive closure of the pairs we have generated so far. Two terms that are equivalent in this relation are conjectured to be equivalent in $T$ based on a sampling of their possible values. Now, let $\{r_1, \ldots, r_n\} \subset D$ be a set of representative terms from this equivalence relation, that is, this set contains exactly one term from each equivalence class. When a new term $t$ is added to our term database $D$, we either (a) determine that $t$ is equivalent to some $r_i$, output $t \approx r_i$ as a candidate rewrite rule and add $t$ to the equivalence class of $r_i$, or (b) determine that $t$ is not equivalent to any of $r_1, \ldots, r_n$ and add $\{t\}$ to our set of equivalence classes. We discuss below the criteria we use to determine this equivalence via sampling methods, where two terms $t_1$ and $t_2$ are considered equivalent if they rewrite to the same value for some list of value assignments to their variables. In other words, $t_1$ and $t_2$ are equivalent iff $[(t_1[\bar{y} \mapsto \bar{c}_1])\downarrow, \ldots, (t_1[\bar{y} \mapsto \bar{c}_n])\downarrow] \equiv [(t_2[\bar{y} \mapsto \bar{c}_1])\downarrow, \ldots, (t_2[\bar{y} \mapsto \bar{c}_n])\downarrow]$. We call $\bar{c}_1, \ldots, \bar{c}_n$ sample points. Our equivalence checking criteria can thus be characterized by different choices for constructing a set $P$ of sample points.

**Random Sampling** A naïve method for constructing $P$ is to choose $N$ points at random. To do so, we have implemented a random value generator for each of the types $\tau$ we are interested in. For some types such as $\text{Bool}$ and fixed-width bit-vectors $\text{Bv}$, this generator is straightforward: it returns a uniformly random value in the (finite) range. For other types like the integers $\text{Int}$ and the character strings $\text{Str}$, we apply a recursive procedure that chooses a random digit/character and subsequently either repeats or terminates with some probability (0.5 in our current implementation). For the reals $\text{Real}$, we return the rational $c_1/c_2$ where $c_1$ is an integer and $c_2$ a non-zero integer chosen at random from some large interval.

**Grammar-Based Sampling** While random sampling is very easy to implement, it is not very effective at generating points that witness the non-equivalence of certain term pairs. For this reason, we have implemented an alternative method that constructs points based on the user-provided grammar. In contrast with random sampling, we construct a set $P$ of $N$ points by computing $\langle(t_1[\bar{y} \mapsto \bar{c}_1])\downarrow, \ldots, (t_n[\bar{y} \mapsto \bar{c}_n])\downarrow\rangle$ where $\bar{c}_1, \ldots, \bar{c}_n$ are random points, and for $i = 1, \ldots, n$, the term $t_i$ is generated by the grammar $R = (s_0, S, R)$ starting from some symbol $s \in S$ such that $s \rightarrow t_i \in R$. The intuition here is that sample points of this form are biased towards interesting values. In particular notice that points of this form are highly likely to include combinations of the user-provided constants that occur in the input grammar.

**Exact Checking with Model-based Sampling** In contrast to the previous two methods this one is exact; that is, it makes two terms equivalent only if they are $T$-equivalent. It is based on satisfiability checking and dynamic, model-based generation of our sample points $P$. The method maintains a growing set of sample points $P$, initially empty. When a term $t$ is generated, we check if it agrees on the current set of sample points $P$ with any previously generated term $s$. If so, we separately check the $T$-satisfiability of $\exists \bar{y} t \neq s$ where $\bar{y}$ includes the free variables of $s$ and of $t$. If this query is unsatisfiable, then $t \approx s$ is guaranteed to be a correct rewrite rule and we output it. If instead, $t \neq s$ is satisfied by some model $M$, we output $t \approx s$.

---

3 This is a slight misnomer since we do not insist on an orientation between $t_i$ and $s_i$.
4 Typically, the representative of an equivalence class is the first term from the class that was added to $D$. 
then we add the evaluation $M(\bar{y})$ of $\bar{y}$ in the model $M$ as a new sample point to $P$, to be used for the subsequently generated terms. Since, $s$ and $t$ are provably inequivalent in $T$, we return no rewrite rule for this pair.

### 4 Filtering Techniques for Rewrite Rules

An important aspect in the workflow in Figure 1 from a developer’s perspective is that the system suggests not only suitable rewrite rules but useful ones. In particular, it is preferable that the set of candidate rewrite rules do not include trivial consequences of previous rewrite rules. Thus, we have implemented several additional filtering techniques, which we describe in this section. In the following, we say a candidate rewrite rule $t \approx s$ is redundant with respect to set $\{t_1 \approx s_1, \ldots, t_n \approx s_n\}$ if $\forall \bar{y} \ (t_1 \approx s_1 \land \ldots \land t_n \approx s_n)$ entails $\forall \bar{y} \ t \approx s$ in theory $T$, where $\bar{y}$ collects all the free variables in these terms. Checking this entailment directly is impractical, since it involves first-order quantification modulo $T$. However, we have implemented several efficient and sound techniques to prove entailments of this form. These significantly reduce the number of rewrite rules we print in Step 4 of the workflow, as we discuss in Section 5.1.

**Filtering based on Matching** One simple way to infer that a rewrite rule is redundant is to check that it is an instance of a previously generated rule. For example, $y_1 + 1 \approx 1 + y_1$ is clearly redundant with respect to any set that includes $y_1 + y_2 \approx y_2 + y_1$ and $y_1 + y_2 \approx y_2 + y_1$. Our implementation filters rewrite rules like the one above using a discrimination tree data structure over all representative terms generated as a result of our equivalence checking in the previous step. When a new candidate rewrite rule $t \approx s$ is generated, we first add the term $t$ to this data structure. It outputs a set of matches of the form $t_1 \sigma_1, \ldots, t_n \sigma_n$ where for each $i = 1, \ldots, n$, we have that $t_i$ is a previous term added to the structure, and $t_i \sigma_i = t$. If for any such $i$, we have that $t_i \approx s_i$ was a previously generated candidate rewrite rule, and $s_i \sigma_i = s$, we have inferred that $t \approx s$ is an instance of the rule $t_i \approx s_i$ for substitution $\sigma_i$, and hence $t \approx s$ is not reported to the user.

**Filtering based on Congruence** Another simple way to infer that a rewrite rule is redundant is to verify that it can be inferred by congruence closure. For example, it is easy to infer, for any function $f$, that the rule $f(y_1 + 0) \approx f(y_1)$ is redundant with respect to any set that contains $y_1 + 0 \approx y_1$. Our implementation filters candidate rewrite rules like the one above by maintaining a data structure that represents the congruence closure $C(E)$ of the current set $E$ set of rewrite rules. That is, the smallest superset of $E$ that is closed under entailment in the theory of equality and (uninterpreted) function symbol. We discard any new candidate rewrite rule $t \approx s$ if the equality $t \approx s$ is already a consequence of $C(E)$.

### 5 Preliminary Experience

We discuss our implementation of the proposed workflow, evaluate different configurations, and show the impact on benchmarks. We ran all experiments on a cluster equipped with Intel E5-2637 v4 CPUs running Ubuntu 16.04. We provisioned one core and 8 GB RAM for each job.

We have implemented our approach for generating rewrite rules in cvc4, a state-of-the-art SMT solver. In the past few months, we generated rewrite rules using several grammars and started implementing promising candidates in an extended rewriter, ext, that can be enabled optionally. The rewrites implemented in ext are a superset of the rewrites in the default rewriter std. So far, we have used our approach to implement rewrite rules in the theory of strings, the theory of bit-vectors and for Booleans.

In our experience, the rewrite rules suggested by our approach can be very subtle. For example, our approach suggested that `(str.substr "B" z z)` can be rewritten to the empty string (if $z$ is zero, then

---

5 We treat all symbols as uninterpreted functions in this setting
6 We give details on three of these in this section.
Figure 2: Grammars used for comparisons. From left to right: strterm, bvterm (4-bit variant), crci.

For our evaluation, we consider three grammars: strterm for the theory of strings, bvterm for the theory of bit-vectors, and crci for Booleans. For bvterm, we consider two variants, 4-bit (bvterm4) and 32-bit (bvterm32), that use different bit-widths for constants, variables and operations variants to show the impact of the bitwidth. Figure 2 lists the grammars in SyGuS syntax [17]. Each SyGuS problem specifies a function \( f \) with a set of parameters (e.g. two strings \( x \) and \( y \) and an integer \( z \) in the case of strterm) and a return type (e.g. a function returning a string for strterm) to synthesize. The initial symbol of the grammar is always \( \text{Start} \) and the rules of the grammar are given by a symbol (e.g. \( \text{ie} \) in strterm) and a list of terms (e.g. the constants \( 0 \), \( 1 \) and the functions \( \text{str.len}, \text{str.to.int} \) for the grammar symbol \( \text{ie} \)). SyGuS problems generally have semantic constraints for the functions to be synthesized but for our use case we want to enumerate as many terms as possible for a given grammar, so do not provide any, and let our implementation enumerate all solutions. Note that the number of
Table 1: Impact of different rewriters on the amount of redundant terms generated, using grammar equivalence checking. All times are given in seconds. * indicates that the estimate is based on the variant with the lower bit-width.

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Size</th>
<th>Terms</th>
<th>Unique</th>
<th>none</th>
<th>std</th>
<th>ext</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Redundancy</td>
<td>Time</td>
<td>Redundancy</td>
<td>Time</td>
</tr>
<tr>
<td>strterm</td>
<td>1</td>
<td>218</td>
<td>≤83.0%</td>
<td>0.47</td>
<td>17.3%</td>
<td>0.28</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>24587</td>
<td>≥83.0%</td>
<td>352.83</td>
<td>≤49.7%</td>
<td>90.02</td>
</tr>
<tr>
<td>bvterm</td>
<td>1</td>
<td>63</td>
<td>65.1%</td>
<td>0.20</td>
<td>0.0%</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2343</td>
<td>87.7%</td>
<td>4.91</td>
<td>22.0%</td>
<td>0.97</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>110583</td>
<td>95.7%</td>
<td>5714.03</td>
<td>39.3%</td>
<td>96.35</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5865303</td>
<td>≥83.0%</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>bvterm12</td>
<td>1</td>
<td>63</td>
<td>65.1%</td>
<td>0.29</td>
<td>8.3%</td>
<td>0.10</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2343</td>
<td>87.6%</td>
<td>24.90</td>
<td>21.4%</td>
<td>2.44</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>110583</td>
<td>≥83.0%</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>crci</td>
<td>1</td>
<td>4</td>
<td>25.0%</td>
<td>0.22</td>
<td>25.0%</td>
<td>0.22</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>32</td>
<td>62.5%</td>
<td>0.18</td>
<td>52.0%</td>
<td>0.27</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>276</td>
<td>84.1%</td>
<td>1.05</td>
<td>74.4%</td>
<td>0.86</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2656</td>
<td>93.4%</td>
<td>8.88</td>
<td>87.5%</td>
<td>8.79</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>17920</td>
<td>98.7%</td>
<td>132.73</td>
<td>96.9%</td>
<td>81.64</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>107632</td>
<td>99.7%</td>
<td>4127.80</td>
<td>99.0%</td>
<td>1156.01</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>588064</td>
<td>—</td>
<td>—</td>
<td>99.8%</td>
<td>19128.83</td>
</tr>
</tbody>
</table>

Unique Solutions In Table 1, we provide an overview of the number of terms and the number of unique terms modulo $T$-equivalence at different sizes for each grammar. We established the number of unique terms for the Boolean and bit-vector grammars using the exact equivalence checking, and approximated the number of unique terms for the string grammar using grammar with 10,000 samples. We use grammar-based sampling for strings because it is currently not known whether the theory of strings is decidable [9] and the procedure for the theory of strings in cvc4 [14] is not guaranteed to terminate. None of our equivalence checks may result in false negatives, i.e. they will never declare two terms to be different when they are actually equivalent. Thus, the number of unique terms reported by grammar is a lower bound of the actual number of unique terms. For all grammars, the number of terms grows rapidly with increasing size while the number of unique terms grows much slower. As the size increases, the relative number of redundant terms quickly approaches 100% when none is used. This indicates enumerating terms without filtering based on rewriting can be very inefficient for larger sizes, and suggests that there is high potential for rewrite rules in these domains. Interestingly, the number of unique terms differs between the 4-bit and the 32-bit versions of bvterm at size 2. This is due to bit-width dependent rewrites that are valid for smaller bit-widths only.

Rewriter Comparison To measure the impact of the rewriter, we used our three rewriter configurations
Table 2: Comparison of different equivalence checks, the relative number of candidates filtered and the overhead of checking rewrites for soundness, using the ext rewriter. All times are given in seconds.

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Size</th>
<th>no-eqc</th>
<th>random</th>
<th>grammar</th>
<th>exact</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time</td>
<td>Error</td>
<td>Time</td>
<td>Error</td>
</tr>
<tr>
<td>strterm</td>
<td>1</td>
<td>0.23</td>
<td>0.0%</td>
<td>0.22</td>
<td>0.0%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>41.30</td>
<td>≤6.6%</td>
<td>73.13</td>
<td>1.1%</td>
</tr>
<tr>
<td>bvterm(_4)</td>
<td>1</td>
<td>0.06</td>
<td>0.0%</td>
<td>0.13</td>
<td>0.0%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.61</td>
<td>0.0%</td>
<td>0.71</td>
<td>0.0%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>47.84</td>
<td>0.0%</td>
<td>56.21</td>
<td>0.0%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2114</td>
<td>0.0%</td>
<td>26213</td>
<td>0.0%</td>
</tr>
<tr>
<td>bvterm(_{32})</td>
<td>1</td>
<td>0.09</td>
<td>27.3%</td>
<td>0.07</td>
<td>0.0%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.69</td>
<td>62.8%</td>
<td>2.84</td>
<td>15.9%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>42.57</td>
<td>≤79.1%</td>
<td>177.98</td>
<td>38.7%</td>
</tr>
<tr>
<td>crci</td>
<td>1</td>
<td>0.05</td>
<td>0.0%</td>
<td>0.07</td>
<td>0.0%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.07</td>
<td>0.0%</td>
<td>0.07</td>
<td>0.0%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.61</td>
<td>0.0%</td>
<td>0.57</td>
<td>0.0%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>3.08</td>
<td>0.0%</td>
<td>3.25</td>
<td>0.0%</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>15.05</td>
<td>0.0%</td>
<td>15.01</td>
<td>3.3%</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>37.31</td>
<td>0.0%</td>
<td>40.55</td>
<td>0.0%</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>55.63</td>
<td>0.0%</td>
<td>62.56</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

Table 2: Comparison of different equivalence checks, the relative number of candidates filtered and the overhead of checking rewrites for soundness, using the ext rewriter. All times are given in seconds.

For bvterm\(_4\) and crci there is no difference between random and grammar.

and measured their term redundancy\(^7\) for the set of terms up to given sizes for each grammar, as well as the wall-clock time to enumerate and check all the solutions. We used grammar with 1,000 samples for the equivalence checks. Table 1 summarizes the results. Without a rewriter, the redundancy is very high at larger sizes, whereas std keeps the redundancy much lower, except in the case of crci. This is due to the fact that the existing Boolean rewriter in cvc4 only performs basic rewriting because cvc4 relies on a SAT solver to perform Boolean reasoning. As expected, ext fares much better in all cases, lowering the percentage of redundant terms by over 70% in the case of crci at size 5. This has a significant effect on the time it takes to enumerate all solutions: ext consistently and significantly outperforms std, in some cases by more than five times, especially at larger sizes. Compared to none both std and ext perform much better.

**Equivalence Check Comparison** To compare the different equivalence checks, we measured their error\(^8\) with respect to the set of terms enumerated by the ext rewriter, and the wall-clock time to enumerate all the solutions. For both random and grammar, we used 1,000 samples. We summarize the results in Table 2. For crci and the 4-bit variant of bvterm, 1,000 samples are enough to cover all possible inputs, so there is no error in those cases. For both variants of bvterm\(_4\) and crci, there is no difference between random and grammar, which is why we only report the former. While sampling performs similarly to exact for crci and the 4-bit variant of bvterm, for the 32-bit variant, it is significantly more precise and slightly faster for size 2. At size 3, exact consumed more than 8 GB RAM and was killed. For strterm, we found that the use of grammar-based sampling was much more effective than random sampling, having an error that was around 6 times smaller (≤1.1%) for depth 2. Recall that since we did not establish the number of unique terms using exact for this depth, this error is relative to grammar with 10,000 samples.

\(^7\) We define the term redundancy of a rewriter (\(\downarrow\)) with respect to a (non-empty) set of terms \(S\) to be \((n - u)/n\), where \(n\) is the cardinality of \(\{t\mid t \in S\}\) and \(u\) is the number of unique terms in \(S\) (the size of a maximal subset of \(S\) whose terms are pairwise \(T\)-disequivalent), where notice that \(n ≥ u\).

\(^8\) Similarly, we define the error of an equivalence check with respect to a (non-empty) set of terms \(S\) to be \((u - n)/u\), where \(n\) is number of equivalence classes of \(S\) induced by the check and \(u\) is the number of unique terms in \(S\), where notice that \(u ≥ n\).
Figure 3: Impact of new rewrites for SyGuS benchmarks over Strings, Bitvector and Boolean logics. All benchmarks run with a 3600 second timeout.

Thus, these numbers also indicate that a relatively number of samples using `grammar` is good enough to discard a majority of spurious rewrites. As expected, `exact` gets stuck and times out for `strterm` at size 2. The first difficult query is proving the equivalence of `(str.replace x (str.replace x "B" x) x)` and `x`. We plan to collect these problems that are challenging to the current decision procedure and use them to improve the rewriter as well as the decision procedure itself.

**Impact of Filtering** In Table 2, we report how many candidate rewrites were filtered using our filtering techniques described in Section 4. We used `ext` and `exact` if available and `grammar` otherwise. Our filtering techniques eliminate up to $64\%$ of the rules, which significantly lowers the burden on the developer when analyzing the proposed rewrite rules.

### 5.2 Improving Confidence in the Rewriter

Soundness is of utmost importance when implementing a rewriter in an SMT solver. An unsound rewriter often implies that the overall SMT solver is unsound. To accommodate the rapid development of new rewrite rules in our workflow in Figure 1, we have instrumented an optional mode of `cvc4` that attempts to detect unsoundness in the implementation of its current rewriter. When this mode is enabled, for each term $t$ enumerated in Step 2, we use the grammar-based technique from Section 3 to test the equivalence of $t$ and $t\downarrow$. This functionality has been critical for catching subtle bugs in implementation of new rewrite rules. In Table 2, we provide a column that indicates the overhead of running this checking for each grammar and term size, where grammar-based sampling is used both for computing the rewrite rules and for checking the soundness of the `ext` rewriter. For example, adding checks that ensure that no unsound rewrites are produced for terms up to size 3 in the `bvterm_{32}` grammar has a 341.7% overhead, or is roughly four and a half times slower. Overall, this option leads to a noticeable, but not prohibitively large, slowdown in the run-time of the overall system.

### 5.3 Impact of Rewrites on Solving

We give a brief evaluation of the impact of our newly developed rewrites have had on solving times of `cvc4`. We focus on two categories: syntax-guided synthesis problems and traditional SMT problems.

**Syntax-guided Synthesis** In Figure 3, we give three scatter plots that compare the syntax-guided synthesis solver of `cvc4` that is instrumented to use the extended rewriter developed as the result of

---

* A rewriter is unsound if there exists a pair of $T$-disequivalent terms $t$ and $s$ such that $t\downarrow = s\downarrow$. 

this work (ext), and the rewriter from version 1.5 (std). All plots use a logarithmic scale. The string problems come from a programming-by-examples application [10] and the Boolean problems come from a cryptographic circuit synthesis application [8]. For these two, the grammars are similar to those shown in Figure 2. The bit-vector problems cover a range of applications [12, 1, 13], where the grammars vary significantly. We observe that the performance of cvc4’s enumerative syntax-guided synthesis solver is highly related to the speed at which it is able to enumerate terms that are unique up to equivalence. Concretely, this means that solving time in this domain is directly proportional to the time columns from Table 1 for the various rewriters and sizes. Overall, on the benchmarks that both configurations of cvc4 solved, the use of the extended rewriter was on average a factor of 1.54 times faster on the strings benchmarks, a factor of 1.08 faster on the bit-vector benchmarks and a factor of 2.96 times faster on the Boolean benchmarks. Additionally, the use of the extended rewriter enabled us to solve 4 more bit-vector problems and 24 more Boolean problems within a 3600 second timeout.

SMT Solving While the above evaluation shows that improving a rewriter has a clear positive impact on syntax-guided synthesis solving, our updates to the rewriter in cvc4 so far have a mixed impact on general SMT solving. We considered 25421 strings benchmarks corresponding to symbolic execution of Python programs [21], 40043 benchmarks in the quantifier-free bit-vector logic of SMT-LIB [5] (QF_BV), and 5151 benchmarks in the quantified bit-vector logic (BV). We used a 30 second timeout for the first set and a 300 second timeout for the other two. For the first two sets, cvc4 with the extended rewriter ext had a positive impact on unsatisfiable benchmarks where it was (+12, -1) on the strings set and (+232,-158) on QF_BV, but a negative impact on satisfiable benchmarks where it was (+13, -94) on the strings set and (+143, -236) on QF_BV. For the quantified bit-vector set, cvc4 with the extended rewriter has an overall positive impact for both satisfiable and unsatisfiable benchmarks, where it was a combined (+42,-15) with respect to cvc4 with its default rewriter. We believe this indicates the importance of selecting a subset of possible rewrites to be enabled based on the application.

6 Related Work

A number of techniques have been proposed for automatically generating rewrite rules for bit-vectors. For some examples, SWAPPER [24] is an automatic formula simplifier generator based on machine learning and program synthesis techniques. In the context of symbolic execution, Romanoe et al. [22] propose an approach that learns rewrite rules to simplify expressions before sending them to an SMT solver. In contrast to these works, our approach targets the developer of the SMT solver itself. A related approach was explored by Hansen [11], which generates all the terms that fit a grammar and finds equivalent pairs that can be used by the developer to implement new rules. In contrast to our work, the candidate rules are not filtered, the grammar is hard-coded and only considers bit-vector operations. Other work by Nadel [15] proposes generating bit-vector rewrite rules in SMT solvers at runtime for a given problem. Syntax-guided synthesis was used by Niemetz et al. [16] to synthesize conditions that characterize when bit-vector constraints have solutions. Rewrite rules in SMT solvers—especially the ones for the theories of bit-vectors and Booleans—bear similarities with local optimizations in compilers [3, 6]. Finally, caching counterexamples as we do in our exact equivalence check is similar to techniques used in symbolic execution engines, e.g. KLEE [7], and superoptimizers, e.g. STOKE [23].

7 Conclusion

We have presented a paradigm for rewrite rule development in SMT solvers and an encouraging report of our preliminary experience with it. In ongoing work, we plan to (partially) automate the implementation of the rewriter in Step 5 by automatically inferring optimal configurations that enable or disable specific
rewrites that the developer implements. This is a key component towards developing a rewriter that is optimal for SMT constraints in applications. We are also working on providing an interface for external users who are interested in enumerating rewrite rules relative to a user-provided rewriter.

References


A.1 strterm

Listing 1: none

A Example Output

We provide sample output of cvc4 in its rewrite rule enumeration mode. For each grammar, we give the results of cvc4 with no rewriter (none), its rewriter from cvc4 version 1.5 (std), and the rewriter we generated as a result of this work so far (ext). In each case, the output is a list of unoriented pairs indicating possible rewrites. All variables x, y, z, w, s, t, ... should be interpreted as universal, in that the rewrite holds for all values of x, y, z, w, s, t, .... Intuitively, the output when using none corresponds to a list of rewrites over terms in the grammar if the developer was writing a rewriter from scratch; the output when using std (resp. ext) corresponds to the simplest rewrites, measured in term size, over the terms from the grammar that the given rewriter currently cannot infer.
Listing 2: std

 חדר 2: std

 Listing 3: ext

 חדר 3: ext

A.2 bvterm

A.2 bvterm

Listing 4: none

 חדר 4: none

Listing 5: std

 חדר 5: std

Listing 6: ext

 חדר 6: ext
A.3 crci

Listing 7: none

(candidate-rewrite (or x x) (and x x))
(candidate-rewrite (and (or w w) x) (and (and w w) x))
(candidate-rewrite (and (xor w w) x) (xor x x))
(candidate-rewrite (and x (and w w)) (and (and w w) x))
(candidate-rewrite (and x (not w)) (and (not w) x))
(candidate-rewrite (and x (or w w)) (and (and w w) x))
(candidate-rewrite (not (or w w)) (not (and w w)))
(candidate-rewrite (or (or w w) x) (or (and w w) x))
(candidate-rewrite (or (not w)) (or (not w) x))
(candidate-rewrite (or (xor w w) x) (and x x))
(candidate-rewrite (or (or w w) x) (and x x))

Listing 8: std

(candidate-rewrite (or x x) (and x x))
(candidate-rewrite (and (or w w) x) (and (and w w) x))
(candidate-rewrite (and x (not w)) (and (not w) x))
(candidate-rewrite (and x (or w w)) (and (and w w) x))
(candidate-rewrite (not (or w w)) (not (and w w)))
(candidate-rewrite (or (or w w) x) (or (and w w) x))
(candidate-rewrite (or (not w)) (or (not w) x))
(candidate-rewrite (or (xor w w) x) (and x x))
(candidate-rewrite (or (or w w) x) (and x x))

Listing 9: ext

(candidate-rewrite (xor x (and (and y y) w)) (xor (and (and y y) w) x))
(candidate-rewrite (xor x (or (and y y) w)) (xor (or (and y y) w) x))
(candidate-rewrite (xor x (or (not y) w)) (xor (or (not y) w) x))
(candidate-rewrite (and (xor w (and (and z z) y) w) (xor (and (and z z) y) w) x))
(candidate-rewrite (and (xor w (and (not z) y) w) (xor (and (not z) y) w) x))
(candidate-rewrite (and (xor w (or (and z z) y) w) (xor w (or (and z z) y) w) x))
(candidate-rewrite (and (xor w (or (not z) y) w) (xor w (or (not z) y) w) x))
(candidate-rewrite (or (xor w (and (and z z) y) w) (xor (and (and z z) y) w) x))
(candidate-rewrite (or (xor w (and (not z) y) w) (xor (and (not z) y) w) x))