Designing Theory Solvers with Extensions

Andrew Reynolds[1], Cesare Tinelli[1], Dejan[1] Jovanović[3], and Clark Barrett[2]

[1] Department of Computer Science, The University of Iowa
[2] Department of Computer Science, Stanford University
[3] SRI International

Abstract. Satisfiability Modulo Theories (SMT) solvers have been developed to natively support a wide range of theories, including linear arithmetic, bit-vectors, strings, algebraic datatypes and finite sets. They handle constraints in these theories using specialized theory solvers. In this paper, we overview the design of these solvers, specifically focusing on theories whose function symbols are partitioned into a base signature and an extended signature. We introduce generic techniques that can be used in solvers for extended theories, including a new context-dependent simplification technique and model-based refinement techniques. We provide case studies showing our techniques can be leveraged for reasoning in an extended theory of strings, for bit-vector approaches that rely on lazy bit-blasting and for new approaches to non-linear arithmetic.

1 Introduction

A growing number of formal methods applications leverage SMT solvers as reasoning engines. To accommodate the unique requirements of these applications, a number of new theories are now natively supported by SMT solvers, including unbounded strings with length constraints [39, 31], algebraic datatypes [33], finite sets [5], and floating-point arithmetic [13]. Solvers for these theories share functionalities, such as reporting conflicts and propagations based on theory reasoning. From both a formal and an engineering perspective, there is a need to express the common features in these solvers.

This paper focuses on theories whose function symbols can be partitioned into a *base* signature Σ^{b} and an *extension* signature Σ^{e}. We will refer to such theories as *extended* theories. The motivation for considering extended theories is two-fold:

1. Assume we have developed a constraint solving procedure for some Σ^{b}-theory, and say we want to extend this procedure to handle additional symbols in some signature Σ^{e}. Can we reuse our procedure for Σ^{b}-constraints in part to develop a procedure for $\Sigma^{\mathrm{b}} \cup \Sigma^{\mathrm{e}}$-constraints?
2. Assume we want to optimize a procedure for Σ-constraints. One way is to partition its signature Σ into $\Sigma^{\mathrm{b}} \cup \Sigma^{\mathrm{e}}$, where Σ^{b} contains the symbols that are easier to reason about. Can we use a stratified approach that first uses our existing procedure on Σ^{b}-constraints and reasons about Σ^{e}-constraints only when needed?

We develop an approach for handling extended theories can be used for answering both of these questions. This paper observes that the design of many theory solvers for extended theories follows a similar pattern. First, we observe that it is often possible to

reduce extended constraints to basic ones by reasoning modulo the equalities entailed by the current assignment. As a simple example, in the context where $y \approx 2$ is entailed by the current assignment, the non-linear constraint $x \times y + y > 5$ can be simplified to a linear one $2 \times x > 3$. We refer to this technique as *context-dependent simplification*. Constraints that are not reducible in this way can be handled by techniques that follow the common paradigm of model-based abstraction refinement, where basic constraints can be used to refine the abstraction of extended terms. The latter is an approach followed used by several recent approaches to SMT solving [15, 17].

In previous work, we showed that techniques based on simplification can significantly improve the performance of DPLL(T)-based string solvers [34]. In this work: we formalize the design of theory solvers with extensions, specifically:

- we introduce a generic technique, which we call *context-dependent simplification*, which can reduce extended constraints to basic ones and propagate equalities between extended terms;
- we define a generic approach for extended theories that leverages this technique and others to implement modular extensions for the theories of strings, linear arithmetic and bit-vectors, showing that:
  - context-dependent simplification techniques significantly improve the performance and precision of our solver for an extended theory of strings;
  - lightweight techniques based on context-dependent simplification and model-based refinement can extend DPLL(T) linear arithmetic solvers to handle non-linear arithmetic and have some advantages over state-of-the-art solvers; and
  - the performance of bit-vector solvers can be improved by delaying bit-blasting of certain functions that require sophisticated propositional encodings.

1.1 Formal preliminaries

We assume the reader is familiar with the following notions from many-sorted logic with equality: (sorted) signature, term, literal, formula, clause, free variable, interpretation, and satisfiability of a formula in an interpretation (see, e.g., [11] for more details). We consider only signatures Σ that contain an (infix) logical symbol $\approx$ for equality. We write $t \not\approx s$ as shorthand for $\neg t \approx s$. We write $\mathcal{L}it(\varphi)$ to denote the set of literals of formula φ. We extend these notations to tuples and sets of terms or formulas as expected.

If φ is a Σ-formula and $\mathcal{I}$ a Σ-interpretation, we write $\mathcal{I} \models \varphi$ if $\mathcal{I}$ satisfies φ. If t is a term, we denote by $\mathcal{I}(t)$ the value of t in $\mathcal{I}$. A *theory* is a pair $T = (\Sigma, \mathbf{I})$ where Σ is a signature and $\mathbf{I}$ is a class of Σ-interpretations, the *models* of T, that is closed under variable reassignment (i.e., every Σ-interpretation that differs from one in $\mathbf{I}$ only in how it interprets the variables is also in $\mathbf{I}$). A Σ-formula φ is *satisfiable* (resp., *unsatisfiable*) *in* T if it is satisfied by some (resp., no) interpretation in $\mathbf{I}$. A set Γ of Σ-formulas *entails in* T a Σ-formula φ, written $\Gamma \models_T \varphi$, if every interpretation in $\mathbf{I}$ that satisfies all formulas in Γ satisfies φ as well. Two Σ-formulas are *equisatisfiable in* T if for every model $\mathcal{A}$ of T that satisfies one, there is a model of T that satisfies the other and differs from $\mathcal{A}$ at most over the free variables not shared by the two formulas. We say that Γ *propositionally entails* φ, written $\Gamma \models_{\mathrm{p}} \varphi$, if Γ entails φ when considering all atoms as propositional variables.

$\mathsf{Solve}_T(\mathsf{M})$: Return one of the following:
$\mathsf{Learn}(\varphi)$ where $\varphi = \ell_1 \vee \ldots \vee \ell_n,\ \ell_1, \ldots, \ell_n \subseteq \mathcal{L},\ \emptyset \models_T \varphi,$ and $\mathsf{M} \not\models_{\mathrm{P}} \varphi$
$\mathsf{Infer}(\ell)$ where $\mathsf{M} \models_T \ell,\ \ell \notin \mathsf{M},$ and $\ell \in \mathcal{L}$
$\mathsf{Sat}(\mathcal{M})$ where $\mathcal{M} \models \mathsf{M}$
$\mathsf{Unknown}$

Fig. 1. Basic functionality of a theory solver.

2 Theory Solvers

In this paper, we are interested in the design of *theory solvers*. At an abstract level, a theory solver for a Σ-theory T is a terminating procedure specialized in determining the satisfiability of sets of T-literals, interpreted conjunctively. For our purposes, we summarize the interface for a theory solver in Figure 1. We view a theory solver as a procedure Solve_T that takes as input a set of T-literals M, which we will call a *context*, and outputs a value of the following algebraic datatype

$$\mathsf{type}\ \mathit{Response} = \mathsf{Learn\ of}\ \mathit{Clause} \mid \mathsf{Infer\ of}\ \mathit{Literal} \mid \mathsf{Sat\ of}\ \mathit{Model} \mid \mathsf{Unknown}$$

where Clause, $\mathit{Literal}$ and Model are types respectively for representing clauses, literals and interpretations. If $\mathsf{Solve}_T(\mathsf{M}) = \mathsf{Sat}(\mathcal{M})$ then $\mathcal{M}$ is a finitary representation of a model of T that satisfies M, hence we will identity the two in the rest of the paper. We assume that no input context contains both a literal and its negation.

The value returned by Solve_T can be used in various ways depending on the overall search procedure. In most SMT solvers, this search procedure is based on variants of the DPLL(T) procedure [32] where a theory solver for T is used in combination with a CDCL propositional satisfiability (SAT) solver to determine the satisfiability in T of quantifier-free formulas. In a nutshell, given a quantifier-free formula φ, this procedure maintains a set of Σ-clauses F equisatisfiable in T with φ, and tries to construct a context M that is satisfiable in T and propositionally entails F. Such context, if it exists, is a witness of the satisfiability of φ in T. Constructing M and checking its satisfiability in T is done with the aid of a theory solver Solve_T.

As indicated in Figure 1, calling a theory solver on a set M of literal may produce one of four results. In the first case (Learn), the theory solver returns a *lemma*, clause φ that is valid in T and not propositionally entailed by M. This clause may consist of complements of literals in M, indicating that M is unsatisfiable in T, or may contain atoms not in M, indicating to the rest of the DPLL(T) procedure that M needs to be extended further. In the second case (Infer), the theory solver returns a literal ℓ that is entailed by the current context M. We assume here that the literals returned by these calls are taken from a set $\mathcal{L}$ of T-literals that ultimately depends on the original input formula φ. In DPLL(T), this typically includes all literals over the atoms occurring in F, but may include additional ones, for instance, for theory solvers that implement the splitting-on-demand paradigm [10]. In the third case (Sat), the procedure returns a (finitary representation) of a model of T that satisfies M. In the last case, the theory

solver simply returns Unknown, indicating that it is unable to determine the satisfiability of M or suggest further extensions.

Using previous results on DPLL(T) [32, 10]), it can be shown that a DPLL(T) procedure invoking a theory solver Solve_T based on this interface is:

- refutation-sound (i.e., it says an input formula is unsatisfiable in T only if it is so),
- model-sound (i.e.,, it says an input formula is satisfiable in T only if it is so),
- refutation-complete (i.e., it says an input formula is unsatisfiable in T whenever it is so) if Solve_T never returns Unknown, and
- terminating if $\mathcal{L}$ is a finite set.

3 Theory Solvers with Extensions

In this section, we consider a Σ-theory T whose signature Σ is the union $\Sigma^{\mathrm{b}} \cup \Sigma^{\mathrm{e}}$ of a basic signature Σ^{b} and an extention signature Σ^{e} where Σ^{b} and Σ^{e} have the same sort symbols and share no function symbols. We will refer to the function symbols in Σ^{b} as *basic* function symbols, and to those in Σ^{e} as *extension* function symbols.

We are interested in developing a procedure for the T-satisfiability of a set F of Σ-clauses based on the availability of a theory solver $\mathsf{Solve}^{\mathrm{b}}_T$, which implements the interface from Figure 1, for contexts M consisting of Σ^{b}-literals only. For the purposes of the presentation, we assume that the variables in F are are from some infinite set X and we associate to every Σ-term t over X a unique variable z_t not from X which we call *the purification variable for* t. If e is a Σ-term or formula possibly containing purification variables, we denote by $\mathsf{X}(e)$ the set $\{z_t \approx t \mid z_t \text{ is a purification variable in } e\}$; we write $\lceil e \rceil$ to denote the expression $e\sigma$ where σ is the substitution $\{z_t \mapsto t \mid x_t \approx t \in \mathsf{X}(e)\}$. We extend these notations to sets of terms or formulas as expected.

Without loss of generality, we assume every extension function symbol f in F occurs only in terms of the form $f(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ are variables from X. We let $\lfloor \mathsf{F} \rfloor$ be the result of replacing every term t of this form in F by its purification variable z_t. It is not difficult to show that $\lfloor \mathsf{F} \rfloor \cup \mathsf{X}(\lfloor \mathsf{F} \rfloor)$ is equisatisfiable with F in T.

Example 1. Assume $f \in \Sigma^{\mathrm{e}}$. Let F be the set $\{f(x_5, x_3) \approx x_4,\ x_5 \approx f(x_1, x_2)\}$. After replacing $f(x_5, x_3)$ and $f(x_1, x_2)$ with their respective purification variables z_1 and z_2, say, we get $\lfloor \mathsf{F} \rfloor = \{z_1 \approx x_4,\ x_5 \approx z_2\}$ and $\mathsf{X}(\lfloor \mathsf{F} \rfloor) = \{z_1 \approx f(x_5, x_3),\ z_2 \approx f(x_1, x_2)\}$. Note that $\lceil \lfloor \mathsf{F} \rfloor \rceil = \mathsf{F}$. □

We are interested in developing *extended theory solvers* which take as input *extended contexts*, that is, sets of literals of the form $\mathsf{M} \cup \mathsf{X}(\mathsf{M})$, where M a given set of Σ-literals possibly with purification variables (coming from the purification process for F). We discuss in the following two generic classes of techniques: context-dependent simplification and model-based refinement that can be used to develop extended theory solvers on top of a basic solver.

3.1 Context-Dependent Simplification

We first observe that many theory solvers already have several features of interest handling extended contexts $\mathsf{M} \cup \mathsf{X}(\mathsf{M})$, namely they:

1. Compute an equivalence relation over terms $\mathcal{T}(\mathsf{M})$, where t_1 and t_2 are in the same equivalence class if and only if $\mathsf{M} \models_T t_1 \approx t_2$, and
2. Make use of *simplified* forms $t\!\downarrow$ of Σ-terms t, where $\emptyset \models_T t \approx t\!\downarrow$.

Regarding the first point, a number of theory solvers [31, 26, 33, 5] are developed as modular extensions of the standard congruence closure algorithm, which builds equivalence classes over the terms in the current context.

Regarding the second point, computing simplified forms for T-literals is advantageous since it reduces the number of cases that must be handled by the procedure for T. Moreover, it reduces the number of unique theory literals for a given input, which is highly beneficial for the performance of DPLL(T)-based solvers since it allows the underlying SAT solver to abstract multiple T-literals as the same propositional (Boolean) variable. For example, assuming $(x \times 2 > 8)\!\downarrow$ is $x > 4$, the set $\{x \times 2 > 8,\ \neg(x > 4)\}$ can be simplified to $\{x > 4,\ \neg(x > 4)\}$, which is already unsatisfiable at the propositional level. In most SMT solvers, this is determined by simplification and does not require invoking a theory solver that implements a procedure for arithmetic.

We argue that it is helpful to apply the same simplification technique while taking into account the equalities that are entailed by M. In detail, let $\boldsymbol{y}$ be a tuple of variables and $\boldsymbol{s}$ be a tuple of terms from $\mathcal{T}(\mathsf{M})$ where $\mathsf{M} \models_T \boldsymbol{y} \approx \boldsymbol{s}$. Let σ be the substitution $\{\boldsymbol{y} \mapsto \boldsymbol{s}\}$ which we will refer to as a *derivable substitution (in* M*)*. For any term t, we have that $\mathsf{M} \models_T t \approx (t\sigma)\!\downarrow$ by definition of simplifications and derivable substitutions.

Reducing Extended Terms to Basic Terms We may derive equalities between extended terms and basic ones based on simplification. In particular, consider an equality $x \approx t$ from the $\mathsf{X}(\mathsf{M})$ component of our context, recalling that t is a Σ^{e}-term. If $(t\sigma)\!\downarrow$ is a Σ^{b}-term, then it must be that $\mathsf{M} \models_T (x \approx t) \Leftrightarrow (x \approx (t\sigma)\!\downarrow)$. Hence, we may discard $x \approx t$ and handle $x \approx (t\sigma)\!\downarrow$ using the basic procedure.

Example 2. Consider the extended theory A of (integer or rational) arithmetic, whose basic signature $\Sigma^{\mathrm{b}}_{\mathsf{A}}$ contains the symbols of linear arithmetic and whose extension signature $\Sigma^{\mathrm{e}}_{\mathsf{A}}$ contains the multiplication symbol $\times$. Let $\mathsf{M} = \{z \approx x,\ y \approx w + 2,\ w \approx 1\}$ and $\mathsf{X}(\mathsf{M}) = \{x \approx y \times y\}$. Since $\mathsf{M} \models_{\mathsf{A}} y \approx 3$, the substitution $\sigma = \{y \mapsto 3\}$ is a derivable substitution in M. Assuming the simplified form *linearizes* multiplication by constants, we have that $(y \times y)\sigma\!\downarrow\, = (3 \times 3)\!\downarrow\, = 9$ where, observe, 9 is a $\Sigma^{\mathrm{b}}_{\mathsf{A}}$-term. Thus, we may infer the (basic) equality $x \approx 9$ which is entailed by M. $\square$

Inferring Equivalence of Extended Terms If two extended terms t_1 and t_2 can be simplified to the same term under a derivable substitution, we can conclude that they must be equivalent. This is regardless of whether their simplified form is a basic term or not.

Example 3. Let $\mathsf{M} = \{x_1 \not\approx x_2,\ w \approx 4 \cdot z,\ y \approx 2 \cdot z\}$ and $\mathsf{X}(\mathsf{M}) = \{x_1 \approx y \times y,\ x_2 \approx w \times z\}$ where $\cdot$ denotes linear multiplication (i.e., $2 \cdot z$ is equivalent to $z + z$ in A). We have that $\sigma = \{w \mapsto 4 \cdot z,\ y \mapsto 2 \cdot z\}$ is a derivable substitution in M. Moreover, $(y \times y)\sigma\!\downarrow\, = ((2 \cdot z) \times (2 \cdot z))\!\downarrow\, = 4 \cdot (z \times z) = ((4 \cdot z) \times z)\!\downarrow\, = (w \times z)\sigma\!\downarrow$. Thus, we may infer that $x_1 \approx x_2$ is entailed by M (which shows that M is unsatisfiable in A). $\square$

We call this class of techniques *context-dependent simplification*. For theory solvers that build an equivalence relation over terms, a simple method for constructing a derivable substitution is to map every variable in $\mathcal{T}(\mathsf{M})$ to the representative of its equivalence class in the congruence closure of M. However, more sophisticated methods for constructing derivable substitutions are possible, which we will describe later.

3.2 Model-Based Refinement

Note that $\lfloor \mathsf{F} \rfloor$ is effectively a conservative abstraction of F. A complementary approach to context-dependent simplification involves then refining this abstraction as needed to determine the satisfiability of F in T. We do that based on the model that the basic solver finds for a context M, which consists of literals from F. Generally speaking, other SMT theory approaches already rely on some form of model-based refinement [15, 17]. This section defines this notion according to the terminology used here.

Consider an extended context $\mathsf{M} \cup \mathsf{X}(\mathsf{M})$ where context-dependent simplification does not apply, and moreover the basic theory solver has found that M is satisfied by some model $\mathcal{M}$ of T. If $\mathcal{M} \models \mathsf{X}(\mathsf{M})$, then it is a model of our context. On the other hand, if $\mathcal{M} \not\models \mathsf{X}(\mathsf{M})$, then the extended solver may be instrumented to return a clause that when added to F refines the abstraction by eliminating the *spurious* model $\mathcal{M}$. We generate such clauses from *refinement lemmas*.

Definition 1. *Let* $\mathsf{M} \cup \mathsf{X}(\mathsf{M})$ *be an extended context and let* $\mathcal{M}$ *be a model of* T *satisfying* M. *A refinement lemma for* $(\mathsf{M}, \mathsf{X}(\mathsf{M}), \mathcal{M})$ *is a* Σ^{b}*-clause* φ *such that* $\mathsf{X}(\mathsf{M}) \models_T \varphi$ *and* $\mathcal{M} \not\models \varphi$. □

Example 4. Let M be the set $\{x \not\approx 0\}$ and $\mathsf{X}(\mathsf{M})$ be $\{x \approx y \times y\}$. Let $\mathcal{M}$ be a model A satisfying M with $\mathcal{M}(x) = -1$. A refinement lemma for $(\mathsf{M}, \mathsf{X}(\mathsf{M}), \mathcal{M})$ is $x \geq 0$. Observe that $\lceil x \geq 0 \rceil = y \times y \geq 0$ is valid in T. □

An extended solver that constructs a refinement lemma φ for an input context $\mathsf{M} \cup \mathsf{X}(\mathsf{M})$ may return clause $\lceil \varphi \rceil$ which by construction is valid in T, as one can show.

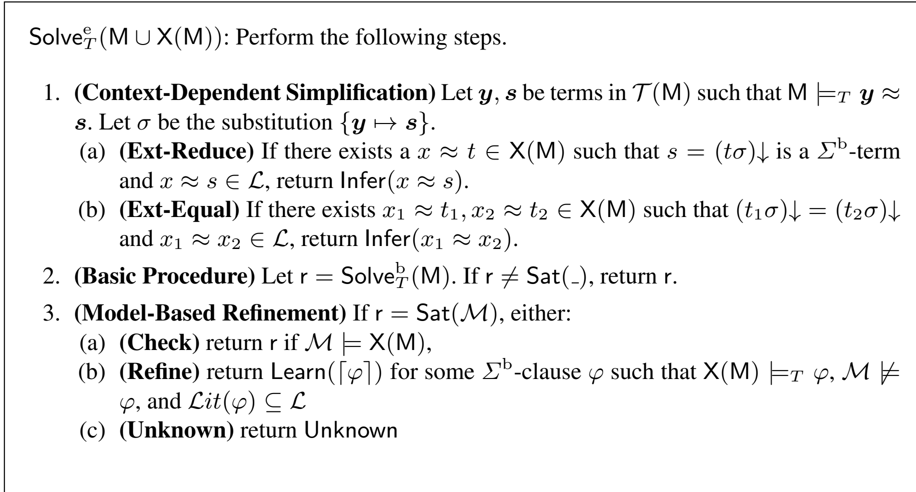
The following definition will be useful when discussing how refinement lemmas are constructed for specific theories.

Definition 2. *Let* $\mathcal{M}$ *be a model of* T*, let* M *a set of basic constraints. The set:*

$$\mathcal{I}^{\mathsf{X}}_{\mathcal{M}}(\mathsf{M}) = \{x \approx t \mid x \approx t \in \mathsf{X}(\mathsf{M}),\ \ell \in \mathsf{M},\ x \in \mathcal{V}(\ell),\ \mathcal{M} \not\models \lceil \ell \rceil\}$$

is the relevant inconsistent subset of $\mathsf{X}(\mathsf{M})$ with respect to $\mathcal{M}$. □

To compute the relevant inconsistent subset of $\mathsf{X}(\mathsf{M})$ with respect to $\mathcal{M}$, we consider each literal $\ell \in \mathsf{M}$, and check whether $\lceil \ell \rceil$ is satisfied by $\mathcal{M}$. For such literal $\ell\sigma, \mathcal{I}^{\mathsf{X}}_{\mathcal{M}}(\mathsf{M})$ contains the equalities $x \approx t$ for purification variables x that occur the free variables of ℓ. Relevant inconsistent subsets are useful because they tell us which variables should likely appear in refinement lemmas.



$\mathsf{Solve}^{\mathrm{e}}_{T}(\mathsf{M} \cup \mathsf{X}(\mathsf{M}))$: Perform the following steps.

1. **(Context-Dependent Simplification)** Let $\boldsymbol{y}, \boldsymbol{s}$ be terms in $\mathcal{T}(\mathsf{M})$ such that $\mathsf{M} \models_T \boldsymbol{y} \approx \boldsymbol{s}$. Let σ be the substitution $\{\boldsymbol{y} \mapsto \boldsymbol{s}\}$.
   (a) **(Ext-Reduce)** If there exists a $x \approx t \in \mathsf{X}(\mathsf{M})$ such that $s = (t\sigma)\!\downarrow$ is a Σ^{b}-term and $x \approx s \in \mathcal{L}$, return $\mathsf{Infer}(x \approx s)$.
   (b) **(Ext-Equal)** If there exists $x_1 \approx t_1, x_2 \approx t_2 \in \mathsf{X}(\mathsf{M})$ such that $(t_1\sigma)\!\downarrow = (t_2\sigma)\!\downarrow$ and $x_1 \approx x_2 \in \mathcal{L}$, return $\mathsf{Infer}(x_1 \approx x_2)$.
2. **(Basic Procedure)** Let $\mathsf{r} = \mathsf{Solve}^{\mathrm{b}}_{T}(\mathsf{M})$. If $\mathsf{r} \neq \mathsf{Sat}(_)$, return r.
3. **(Model-Based Refinement)** If $\mathsf{r} = \mathsf{Sat}(\mathcal{M})$, either:
   (a) **(Check)** return r if $\mathcal{M} \models \mathsf{X}(\mathsf{M})$,
   (b) **(Refine)** return $\mathsf{Learn}(\lceil\varphi\rceil)$ for some Σ^{b}-clause φ such that $\mathsf{X}(\mathsf{M}) \models_T \varphi$, $\mathcal{M} \not\models \varphi$, and $\mathcal{L}it(\varphi) \subseteq \mathcal{L}$
   (c) **(Unknown)** return $\mathsf{Unknown}$

Fig. 2. A strategy for an extended theory solver.

Example 5. Let $\mathsf{M} = \{x \geq 0,\ y \geq 0,\ z \geq 0\}$, $\mathsf{X}(\mathsf{M}) = \{x \approx y \times z\}$, and let $\mathcal{M}$ be the model of T satisfying M where $\mathcal{M}(x) = 3$, $\mathcal{M}(y) = 2$, and $\mathcal{M}(z) = 1$. We have that $\mathcal{I}^{\mathsf{X}}_{\mathcal{M}}(\mathsf{M}) = \emptyset$ since $\lceil x \geq 0 \rceil = y \times z \geq 0$, which is satisfied by $\mathcal{M}$. On the other hand, if M is the set $\{x \geq 3,\ y \geq 0,\ z \geq 0\}$, then $\mathcal{I}^{\mathsf{X}}_{\mathcal{M}}(\mathsf{M}) = \{x \approx y \times z\}$ since $\lceil x \geq 3 \rceil = y \times z \geq 3$ which is not satisfied by $\mathcal{M}$. Intuitively, this means the value of x should be refined based on its definition in $\mathsf{X}(\mathsf{M})$, which is $y \times z$. A possible refinement lemma for $(\mathsf{M}, \mathsf{X}(\mathsf{M}), \mathcal{M})$ is then $(y < 3 \wedge z \approx 1) \Rightarrow x < 3$. □

We will see examples of how refinement lemmas are constructed in Sections 4 through 6, each of which learn Σ^{b}-formulas that state properties of extended terms that appear in the relevant inconsistent subset of the current context.

3.3 A Strategy for Extended Theory Solvers

We summarize a strategy, given by $\mathsf{Solve}^{\mathrm{e}}_{T}$ in Figure 2, for designing a solver to handle an extended theory. It first tries to apply context-dependent simplification techniques based on the two kinds of inferences in Section 3.1. Otherwise, it invokes the basic procedure $\mathsf{Solve}^{\mathrm{b}}_{T}$ on the basic portion M of our context. If this determines that M is satisfied by model $\mathcal{M}$, it uses model-based refinement techniques, as described in Section 3.2. This will either determine that $\mathcal{M}$ is also a model of $\mathsf{X}(\mathsf{M})$ in which case it returns $\mathsf{Sat}(\mathcal{M})$, construct a refinement lemma for $(\mathsf{M}, \mathsf{X}(\mathsf{M}), \mathcal{M})$, or return $\mathsf{Unknown}$. As mentioned, implementations of model-based refinement vary significantly from theory to theory, and hence our definition of how refinement lemmas are chosen is intentionally left underspecified here.

The next three sections considers examples of DPLL(T) theory solvers that are designed according to Figure 2. In each section, we provide details on how the steps

in $\mathsf{Solve}^{\mathrm{e}}_T$ are specifically implemented for that theory. We consider an extended theory of strings, a theory of bit-vectors with a partitioned signature, and the theory of linear arithmetic extended with multiplication.

4 An Efficient Solver for an Extended Theory of Strings

Recently, SMT solvers have been extended with native support for the theory unbounded strings and regular expressions. Implementations of these solvers have significant improved in both performance and reliability in the past several years [39, 31, 1]. This support has enabled a number of applications in security analysis, including symbolic execution approaches that reason about strings as a built-in type [34].

Consider the extended theory of strings whose signature Σ_{S} contains a sort Str for character strings and a sort Int for integers. We partition the function symbols of this signature in two parts. The base signature $\Sigma^{\mathrm{b}}_{\mathsf{S}}$ contains the standard symbols of linear integer arithmetic, words constructed from a finite alphabet $\mathcal{A}$, string concatenation con and string length len. The extension signature $\Sigma^{\mathrm{e}}_{\mathsf{S}}$ contains four function symbols whose semantics are as follows in every model of the theory. For all x, y, z, n, m, the term $\mathsf{substr}(x, n, m)$ is interpreted as the maximal substring of x starting at position n with length at most m, or the empty string if n is an invalid position; $\mathsf{contains}(x, y)$ is interpreted as true if and only if string x contains string y; $\mathsf{idof}(x, y, n)$ is interpreted as the position of the first occurrence of y in x starting from position n, or -1 if y is empty, n is an invalid position, or if no such occurrence exists; $\mathsf{repl}(x, y, z)$ is interpreted as the result of replacing the first occurrence in x of y by z, or x if x does not contain y.

We describe our approach for this extended theory of strings in terms of the three steps outlined in Figure 2.

Procedure for Σ^{b}_{S}-constraints In previous work [31], we developed an efficient calculus for the satisfiability of quantifier-free strings with length constraints. The calculus handles $\Sigma^{\mathrm{b}}_{\mathsf{S}}$-constraints (but not $\Sigma^{\mathrm{e}}_{\mathsf{S}}$-constraints), and also includes partial support for regular expressions. The calculus is implemented as a theory solver in CVC4. At a high level, this solver infers equalities between string variables based on a form of unification (e.g., it infers $x \approx z$ when $\mathsf{con}(x, y) \approx \mathsf{con}(z, w)$ and $\mathsf{len}\, x \approx \mathsf{len}\, z$ are both in M), returns splitting lemmas based on the lengths of string terms and derives conflicts for instance when it can infer an equality between distinct character strings. The decidability of strings constraints, even in the basic signature that includes length constraints, is an open problem [24]. Nevertheless, the calculus from [31] is sound with respect to models and refutations, and terminates often for constraints that occur in applications.

Context-dependent simplification Functions in the extended signature of strings are a clear target for context-dependent simplification, due to the complexity of their semantics and the multitude of simplifications that can be applied to extended string terms. Examples of non-trivial simplifications for extended string terms include:

$$\mathsf{contains}(\mathsf{con}(y, x, \mathsf{abc}), \mathsf{con}(x, \mathsf{a}))\!\downarrow = \top \qquad \mathsf{contains}(\mathsf{abcde}, \mathsf{con}(\mathsf{d}, x, \mathsf{a}))\!\downarrow = \bot$$
$$\mathsf{contains}(\mathsf{con}(\mathsf{a}, x), \mathsf{con}(\mathsf{b}, x, \mathsf{a}))\!\downarrow = \bot \qquad \mathsf{repl}(\mathsf{con}(\mathsf{a}, x), \mathsf{b}, \mathsf{c})\!\downarrow = \mathsf{con}(\mathsf{a}, \mathsf{repl}(x, \mathsf{b}, \mathsf{c}))$$
$$\mathsf{idof}(\mathsf{con}(\mathsf{a}, x, \mathsf{b}), \mathsf{b}, 0)\!\downarrow = 1 + \mathsf{idof}(x, \mathsf{b}, 0) \qquad \mathsf{repl}(x, \mathsf{a}, \mathsf{a})\!\downarrow = x$$

$$
\begin{aligned}
[\![x \approx \mathsf{substr}(y,n,m)]\!] \equiv\ & \mathsf{ite}(\ 0 \le n < \mathsf{len}\, y \wedge 0 < m, \\
& \quad y \approx \mathsf{con}(z_1, x, z_2) \wedge \mathsf{len}\, z_1 \approx n \wedge \mathsf{len}\, z_2 \approx \mathsf{len}\, y \dot{-} m, x \approx \epsilon) \\
[\![x \approx \mathsf{contains}(y,z)]\!] \equiv\ & (x \not\approx \top) \Leftrightarrow \textstyle\bigwedge_{n=0}^{K} n \le \mathsf{len}\, y - \mathsf{len}\, z \Rightarrow \neg [\![z \approx \mathsf{substr}(y, n, \mathsf{len}\, z)]\!] \\
[\![x \approx \mathsf{idof}(y,z,n)]\!] \equiv\ & [\![z_1 \approx \mathsf{substr}(y, n, \mathsf{len}\, y - n)]\!] \wedge \\
& \mathsf{ite}(\ 0 \le n \wedge z \not\approx \epsilon \wedge [\![\top \approx \mathsf{contains}(z_1, z)]\!], \\
& \quad [\![z \approx \mathsf{substr}(z_1, x - n, \mathsf{len}\, z)]\!] \wedge \\
& \quad [\![\bot \approx \mathsf{contains}(\mathsf{substr}(y', 0, x + \mathsf{len}\, z - (n+1)), z)]\!], x \approx -1\ ) \\
[\![x \approx \mathsf{repl}(y,z,w)]\!] \equiv\ & \mathsf{ite}(\ z \not\approx \epsilon \wedge [\![\top \approx \mathsf{contains}(y,z)]\!], \\
& \quad x \approx \mathsf{con}(z_1, w, z_2) \wedge y \approx \mathsf{con}(z_1, z, z_2) \wedge [\![\mathsf{len}\, z_1 \approx \mathsf{idof}(y, z, 0)]\!], \\
& \quad x \approx y\ )
\end{aligned}
$$

Fig. 3. Reduction of Σ_{S}-constraints to $\Sigma_{\mathsf{S}}^{\mathrm{b}}$-constraints for bounded length K, where z_1, z_2 are fresh variables. The operation $n_1 \dot{-} n_2$ denotes the maximum of $n_1 - n_2$ and 0.

The method for computing the simplified form of extended string terms is around 2000 lines of C++ code in the CVC4 code base.[4] Despite the complexity of the simplifier, computing simplified forms often leads to significant performance benefits, as we discuss later. In addition to using aggressive rewriting techniques for extended string terms, it is often advantageous to use methods for constructing derivable substitutions based on flattening sequences of equalities that involve string concatenation terms. For instance, if M contains $x \approx \mathsf{con}(\mathsf{ab}, y)$, $y \approx \mathsf{con}(\mathsf{c}, z)$ and $z \approx \mathsf{con}(\mathsf{de}, u)$, where ab, c and de are string constants, then our implementation computes $\{x \mapsto \mathsf{con}(\mathsf{abcde}, u)\}$ as a derivable substitution in M.

Model-Based Refinement If all string variables are known to have length bounded above by some concrete natural number K, then reasoning about constraints in the full signature Σ_{S} of the extended theory of strings can be reduced to reasoning about $\Sigma_{\mathsf{S}}^{\mathrm{b}}$-constraints. Concretely, for any equality of the form $x \approx f(x_1, \ldots, x_n)$ where $f \in \Sigma_{\mathsf{S}}^{\mathrm{e}}$, we write $[\![x \approx f(x_1, \ldots, x_n)]\!]$ to denote a formula equivalent to $x \approx f(x_1, \ldots, x_n)$ based on the recursive definition in Figure 3. The size of $[\![x \approx f(x_1, \ldots, x_n)]\!]$ is finite since the reduction replaces extended terms with simpler ones based on a well-founded ordering over extended string functions. Our model-based refinement for the extended theory of strings chooses some $x \approx t$ in the relevant inconsistent subset $\mathcal{I}_{\mathcal{M}}^{\mathsf{X}}(\mathsf{M})$ and returns a lemma of the form $(x \approx t) \Leftrightarrow [\![x \approx t]\!]$. The lemmas we learn by this form require us to fix a bound K on the length of strings. Although not shown here, this can be done in an incremental fashion by reasoning about bounded integer quantified formulas, that is formulas of the form $\forall k.\, 0 \le k \le t \Rightarrow \varphi$, where t does not contain k and φ is quantifier-free. Such formulas can be handled in an incomplete way by guessing upper bounds on the value of t, and subsequently applying finite instantiation as needed [34].

Similar techniques are used in a number of approaches to the extended theory of strings [12], which perform this reduction to basic constraints eagerly. In contrast to

[4] See [34] for more details.

	PyEx-c (5557)		**PyEx-z3** (8399)		**PyEx-z32** (11430)		**Total** (25386)	
Solver	#	time	#	time	#	time	#	time
cvc4+sm	**5485**	52m	**11298**	2h33m	**7019**	1h43m	**23802**	5h8m
cvc4+m	5377	1h8m	10355	2h29m	6879	3h6m	22611	6h44m
z3	4695	2h44m	8415	5h18m	6258	3h30m	19368	11h33m
z3str2	3291	3h47m	5908	7h24m	4136	4h48m	13335	16h1m

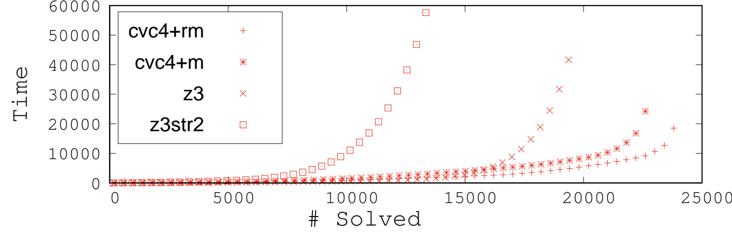


Fig. 4. Table of results of running each solver over benchmarks generated by PyEx, where all benchmarks were run with a 30 second timeout. The cactus plot shows the cumulative runtime taken by each of the four configurations over all benchmarks from the three sets.

those approaches, we perform this reduction in a model-based manner, and only when reasoning by context-dependent simplification does not suffice.

Example 6. Let M be $\{x \approx \bot, y \approx \mathsf{abc}, z \approx \mathsf{con}(\mathsf{b}, w, \mathsf{a})\}$ and $\mathsf{X}(\mathsf{M})$ be $\{x \approx \mathsf{contains}(y, z)\}$, where a, b and abc are string constants. The substitution $\sigma = \{y \mapsto \mathsf{abc}, z \approx \mathsf{con}(\mathsf{b}, w, \mathsf{b})\}$ is a derivable substitution in M. Moreover, $\mathsf{contains}(y, z)\sigma{\downarrow} = \mathsf{contains}(\mathsf{abc}, \mathsf{con}(\mathsf{b}, w, \mathsf{b})){\downarrow} = \bot$ with $\bot$ a basic term. Thus, using context-dependent simplification, we may infer that $x \approx \mathsf{contains}(y, z)$ is equivalent to $x \approx \bot$ in this context. This allows us to avoid constructing the refinement lemma $x \approx \mathsf{contains}(y, z) \Leftrightarrow [\![x \approx \mathsf{contains}(y, z)]\!]$ according to Figure 3. $\square$

Evaluation We considered 25,386 benchmarks generated by PyEx, an SMT-based symbolic execution engine for Python programs which is a recent extension of PyExZ3 [4]. These benchmarks heavily involve string functions in the extended signature. We compare our implementation in the SMT solver CVC4 [7] against Z3-STR [39] and Z3 [19], both of which use eager reductions to handle extended string functions. We tested two configurations of CVC4. The first, **cvc4+m** uses model-based refinement techniques (**m**) for reducing constraints over extended string terms to basic ones. The second, **cvc4+sm** additionally uses context-dependent simplification techniques (**s**) which, following Figure 2, are applied with higher priority than the model-based refinement techniques.[5]

The results are shown in Figure 4 for three sets of benchmarks, **PyEx-c**, **PyEx-z3** and **PyEx-z32**. These benchmarks were generated by PyEx on functions sampled from popular Python packages (httplib2, pip, pymongo, requests) using CVC4, Z3 and Z3-STR as a backend solver respectively. The results show that **cvc4+sm** has better overall performance than the other solvers, solving 23,802 benchmarks while taking a total of

[5] For details on our experiments, see `http://cvc4.stanford.edu/papers/FroCoS2017-ext`.

$$
\begin{array}{rll}
\text{(Sign)} & t_1 \sim_1 0 \wedge t_2 \sim_2 0 \Rightarrow x \sim 0 & \\
\text{(Magnitude)} & |\,t_1\,| \sim_1 |\,s_1\,| \wedge |\,t_2\,| \sim_2 |\,s_2\,| \Rightarrow |\,x\,| \sim |\,(s_1 \times s_2)\,| & \text{where } (s_1 \times s_2)\!\downarrow\, \in \mathcal{T}(\mathsf{X}(\mathsf{M})) \\
\text{(Multiply)} & t_1 \sim_1 p \wedge t_2 \sim_2 0 \Rightarrow x \sim (t_2 \times p) & \text{where } \mathsf{deg}(t_1) \geq \mathsf{deg}(p) \text{ and} \\
 & & (t_1 \sim_1 p)\!\downarrow\, \in \mathsf{M}
\end{array}
$$

Fig. 5. Templates for model-based refinement lemmas for $x \approx t_1 \times t_2$, where t_1, t_2, s_1, s_2 are monomials, p is a polynomial, $\sim_1, \sim_2, \sim \in \{\approx, >, <, \leq, \geq\}$, $|\,t\,|$ is shorthand for the if-then-else term $\mathsf{ite}(t > 0, t, -t)$, and $\mathsf{deg}(t)$ denotes the degree of t.

5 hours and 38 minutes on benchmarks it solves. This is 1,193 more benchmarks that CVC4 with context-dependent simplification disabled, indicating that context-dependent rewriting is a highly effective technique for this set. With respect to its nearest competitor Z3, which took 11 hours and 33 minutes on the 19,368 benchmarks its solves, **cvc4+sm** solved its first 19,368 benchmarks in 1 hour and 23 minutes, and overall solves a total of 4,434 more benchmarks.

5 Lightweight Techniques for Non-Linear Arithmetic

In this section, we consider an extended theory of (real or integer) arithmetic A whose signature Σ_{A} is partitioned so that $\Sigma^{\mathrm{b}}_{\mathsf{A}}$ contains the basic symbols of linear arithmetic, and $\Sigma^{\mathrm{e}}_{\mathsf{A}}$ contains the variadic multiplication symbol $\times$. In the following, a *monomial* refers to a flattened application of multiplication $x_1 \times \ldots \times x_n$, where $x_1, \ldots, x_n$ are (not necessarily distinct) variables. The obvious motivation for this partitioning is that SMT solvers implement efficient decision procedures for linear arithmetic, but their support for non-linear arithmetic is limited (and is necessarily incomplete for integer arithmetic). We outline our approach according to the steps in Figure 2.

Basic Procedure for $\Sigma^{\mathrm{b}}_{\mathsf{A}}$-constraints. Many efficient solvers for linear arithmetic in DPLL(T)-based SMT solvers are based on work by de Moura and Dutertre [22]. Approaches for linear arithmetic in our solver CVC4 are described in King's thesis [29].

Context-dependent simplification. For arithmetic, context-dependent simplification allows us to "linearize" non-linear terms by straightforward evaluation of constant factors. To start, all literals are normalized to atoms of the form $p \sim 0$ where $\sim$ is a relational operator and p is a sum of terms of the form $c \cdot x_1 \times \ldots \times x_n$ with c a concrete integer or rational constant and $x_1 \times \ldots \times x_n$ a monomial. Note a term in this sum is a basic if $m \leq 1$. To construct derivable substitutions for a given set of linear equalities M, we use a technique inspired by Gaussian elimination that finds a set of variables that are entailed to be equal to constants based on the equalities in M. For example, if M contains $x + y \approx 4$ and $y \approx 3$, then $\{x \mapsto 1, y \mapsto 3\}$ is a derivable substitution in M.

Model-Based Refinement. Differently from the theory strings, there is no finite reduction from extended constraints to basic ones for the theory of arithmetic. Instead, our approach for model-based refinement technique for equalities $x \approx t$ in our relevant

inconsistent subset of $\mathsf{X}(\mathsf{M})$, where t is a monomial, adds lemmas that help refine the value of x in future models by stating various properties of multiplication. We see t as decomposed into the product $t_1 \times t_2$ of two monomials. Figure 5 lists three basic templates we use for generating refinement lemmas based on $x \approx t_1 \times t_2$. This list is not comprehensive, but represents the three most commonly used lemma templates in our implementation.

Suppose we have a model $\mathcal{M}$ for our set of basic constraints M. Let φ be a formula that is an instance of one of the templates in Figure 5, meets the side conditions in the figure (if any), and is such that $\lceil \varphi \rceil = \varphi\{x \mapsto t_1 \times t_2\}$ is a valid formula in theory A. Notice that φ is a refinement lemma for $(\mathsf{M}, \mathsf{X}(\mathsf{M}), \mathcal{M})$ if $\mathcal{M} \not\models \varphi$. For the first two lemmas, φ is equivalent to a formula whose literals are either of the form $u_1 \sim u_2$, where $\sim$ is one of $\{\approx, >, <, \leq, \geq\}$, and for $i = 1, 2$, the term u_i is either 0, or a monomial of the form $x_1 \times \ldots \times x_n$, where for each $j = 1, \ldots, n$, x_j is a variable from $\mathcal{V}(\mathsf{X}(\mathsf{M}))$. Only a finite number of literals of this form exist. Thus, all refinement lemmas generated using the first two templates are built from a finite set of literals $\mathcal{L}$. A more detailed argument can show that lemmas generated from the third template are built from a finite set of literals as well. This fact suffices to argue that our extended solver will generate only a finite number of refinement lemmas for a given context M which is enough for termination in DPLL(T). However, it is not enough for refutation completeness in A since one may need refinement lemmas that are not an instance of these templates.

Example 7. Let $\mathsf{M} = \{x < 0, y > z\}$ and $\mathsf{X}(\mathsf{M}) = \{x \approx y \times z\}$. Let $\mathcal{M}$ be a model of M where $\mathcal{M}(x) = -1$, $\mathcal{M}(y) = 3$ and $\mathcal{M}(z) = 2$. The relevant inconsistent subset $\mathcal{I}^{\mathsf{X}}_{\mathcal{M}}(\mathsf{M})$ contains $x \approx y \times z$. The formula $\varphi = (y > 0 \wedge z > 0) \Rightarrow x > 0$ is an instance of first template in Figure 5, and $\lceil \varphi \rceil = (y > 0 \wedge z > 0) \Rightarrow y \times z > 0$ is valid in A. Since $\mathcal{M} \models y > 0 \wedge z > 0$ but $\mathcal{M} \not\models x > 0$, we have that φ is a refinement lemma for $(\mathsf{M}, \mathsf{X}(\mathsf{M}), \mathcal{M})$. Returning $\lceil \varphi \rceil$ as a learned clause has the effect of ruling out a class of models that includes $\mathcal{M}$ in subsequent states. □

Example 8. Let $\mathsf{M} = \{y > 3, x > y, x < 3 \cdot z - 1\}$ and $\mathsf{X}(\mathsf{M}) = \{x \approx y \times z\}$. Let $\mathcal{M}$ be a model of M where $\mathcal{M}(y) = 4$, $\mathcal{M}(x) = 5$ and $\mathcal{M}(z) = 3$, where again $(x \approx y \times z) \in \mathcal{I}^{\mathsf{X}}_{\mathcal{M}}(\mathsf{M})$. The formula $\varphi = (y > 3 \wedge z > 0) \Rightarrow x > 3 \cdot z$ is an instance of the third template in Figure 5, and $\lceil \varphi \rceil = (y > 3 \wedge z > 0) \Rightarrow y \times z > 3 \cdot z$ is valid in A. Since $\mathcal{M} \models y > 3 \wedge z > 0$ but $\mathcal{M} \not\models x > 3 \cdot z$, we have that φ is a refinement lemma for $(\mathsf{M}, \mathsf{X}(\mathsf{M}), \mathcal{M})$. Returning $\lceil \varphi \rceil$ as a learned clause suffices to show this context is unsatisfiable. □

Evaluation We considered all benchmarks of the SMT-LIB library [9] that contain non-linear real (QF_NRA) and non-linear integer (QF_NIA) quantifier-free problems. We evaluated two configurations of CVC4: **cvc4+sm** and **cvc4+m**. The first configuration implements both context-dependent simplification (based on linearizing variables that are entailed to be equal to constants), and model-based refinement lemmas (Figure 5), whereas the second implements model-based refinement only.

The results are presented in Figure 6. On the QF_NRA problems, we compared CVC4 with Z3, YICES2 [21], and RASAT [37]. RASAT is an incomplete interval based

QF_NIA	aprove		calypto		lranker		lctes		leipzig		mcm		uauto		ulranker		Total	
	#	time	#	time	#	time	#	time	#	time	#	time	#	time	#	time	#	time
yices	**8706**	1761	**173**	83	98	102	0	0	92	30	4	32	**7**	0	**32**	11	**9112**	2021
z3	8253	7636	172	146	93	767	0	0	157	173	**16**	180	**7**	0	32	43	8730	8947
cvc4+m	8234	4799	164	43	**111**	52	**1**	0	69	589	0	0	6	0	32	84	8617	5569
cvc4+sm	8190	3723	170	61	108	57	**1**	0	68	375	3	107	7	1	32	86	8579	4413
AProVE	8028	3819	72	110	3	2	0	0	**157**	169	0	0	0	0	6	4	8266	4106

QF_NRA	hong		hycomp		kissing		lranker		mtarski		uauto		zankl		Total	
	#	time	#	time	#	time	#	time	#	time	#	time	#	time	#	time
z3	9	16	**2442**	3903	**27**	443	235	1165	**7707**	370	**60**	175	87	23	**10567**	6098
yices	7	59	2379	594	10	0	213	3110	7640	707	50	210	**91**	61	10390	4744
raSat	20	1	1933	409	12	32	0	0	6998	504	0	0	54	52	9017	999
cvc4+sm	**20**	0	2246	718	5	0	**623**	8375	5434	3711	11	31	33	36	8372	12874
cvc4+m	**20**	0	2236	491	6	0	603	6677	5440	3532	10	33	31	25	8346	10761

Fig. 6. Results for benchmarks in the QF_NIA and QF_NRA logics of SMT-LIB. All experiments are run with a 60 second timeout. Time columns give cumulative seconds on solved benchmarks.

solver, while both Z3 and YICES2 are complete solvers based on NLSAT [28] (with YICES2 relying on the more recent variant called MCSAT [20]). Note that NLSAT and the underlying algorithms are highly non-trivial and not based on DPLL(T), making integration with DPLL(T)-based solvers such as CVC4 impossible.

Although our method is incomplete, overall CVC4 solves an impressive fraction of SMT-LIB problems. The first interesting observation is that CVC4 solves all instances in the **hong** problem set. These are problems that are know to be hard for the methods underlying Z3 and YICES2, but easy for solvers based on interval reasoning such as RASAT. Note that CVC4 does not directly employ any interval reasoning, and the extra deductive power comes as a side-effect of model-based refinement. Another positive result is that CVC4 solves most problems in the **lranker** [30] and **uauto** problem sets. CVC4's perfomance on these problems which come from invariant generation [18], show that our proposed methods work well on practical problems. An example of a class of benchmarks where CVC4 does not perform well are the **mtarski** benchmarks [2]. These benchmarks come from the analysis of elementary real functions and, due to their high degrees, solving them requires full support for algebraic reasoning. The results show that our new method is positioned between the incomplete interval-based methods like those implemented in RASAT, and the complete methods like those implemented in Z3 and YICES2, while performing well on practical problems.

On the QF_NIA problems, we compare CVC4 with Z3, YICES2, and APROVE [25]. The APROVE solver relies on bit-blasting [23], Z3 relies on bit-blasting aided with linear and interval reasoning, while YICES2 extends NLSAT with branch-and-bound [27]. Both versions of CVC4 perform well, especially considering that we do not rely on bit-blasting or sophisticated non-linear reasoning. Again, on the **lranker** and **ulranker** problem sets the new method in CVC4 excels, solving the highest number of problems. Overall, **cvc4+m** proves 812 problems unsatisfiable, and **cvc4+sm** proves 825 problems

unsatisfiable, while YICES2, Z3, and APROVE can show 975, 485 and 0 problems unsatisfiable, respectively. Focusing on unsatisfiable problems, our results show that the new method is positioned between the incomplete bit-blasting-based solvers like APROVE, and more sophisticated solvers like YICES2.

6 Lazy Bit-blasting for Bit-vector Constraints

In this section, we present our preliminary work on a stratified approach for solving bit-vector constraints. We consider the theory of fixed-width bit-vectors whose signature contains a bit-vector sort BV_n for each $n > 0$, and a variety of functions that are used to encode bit-level arithmetic and other operations [8]. A common method for constraints in this theory is to eagerly reduce bit-vector constraints to propositional ones, where this method is often called *bit-blasting*. However, certain bit-vector functions require fairly sophisticated propositional encodings which may degrade the performance of the SAT solver that reasons about the bit-blasted form of the problem. Thus, we consider a theory of bit-vectors whose signature is partitioned such that its extended signature contains the symbols for bit-vector multiplication (bvmul), unsigned and signed division (bvudiv and bvsdiv), unsigned and signed remainder (bvurem and bvsrem), and signed modulus (bvsmod). All other symbols are assumed to be in the basic signature.

Procedure for $\Sigma^{\mathsf{b}}_{\mathsf{BV}}$-constraints In previous work [26], Hadarean et. al. developed lazy techniques for a theory of fixed width bit-vectors. In their approach, the solver resorts to bit-blasting only when algebraic approaches do not suffice to establish satisfiability. The solver may use algebraic reasoning to infer additional equalities, for instance based on specialized reasoning about inequalities, bit-shifting, or concatenation and extraction. If M is still satisfiable, then the solver resorts to bit-blasting. In other words, for each $\ell \in \mathsf{M} \cap \mathcal{L}$, the solver learns the formula $\ell \Leftrightarrow \mathcal{B}(\ell)$, where $\mathcal{B}(\ell)$ is the propositional encoding of bit-vector literal ℓ.

Context-dependent simplification Competitive modern solvers including CVC4 use aggressive simplification techniques for the theory of bit-vectors which we leverage in the first step of Figure 2. Our technique for constructing derivable substitutions is based on mapping variables x to bit-vector constants that occur in the same equivalence class as x in the congruence closure of M.

Model-Based Refinement Our model-based refinement techniques chooses a $x \approx t$ in $\mathcal{I}^{\mathsf{X}}_{\mathcal{M}}(\mathsf{M})$ and learns $x \approx t \Leftrightarrow \mathcal{B}(x \approx t)$, where $\mathcal{B}(x \approx t)$ is the propositional encoding of $x \approx t$. In other words, we bit-blast constraints from $\mathsf{X}(\mathsf{M})$ at lower priority than constraints in M, and only if they appear in our relevant inconsistent subset.

Evaluation We provide a preliminary evaluation of a new version **cvc4+sm** whose signature is partitioned according to this section and that implements both context-dependent simplification and model-based refinement techniques, and compared this with the default configuration of CVC4 with lazy bit-blasting from [26] that does not consider the partitioned signature. We ran both on the **sage2** family of benchmarks

from the QF_BV division of SMT LIB [9]. Overall, **cvc4+sm** solved 11415 benchmarks compared to 11256 solved by **cvc4**. While these results are not competitive with state-of-the-art eager bit-blasting techniques such as those in Boolector [14] which solves 13549, we believe these results are encouraging due to the simplicity of the implementation and orthogonality with eager bit-blasting approaches, as **cvc4+sm** solved 2171 benchmarks in this set not solved by Boolector.

7 Related Work

A common way to support extensions of theories is to provide first-order axiomatizations of additional symbols in the signature of the extension. One can show decision procedures for theory extensions exist, given a finite instantiation strategy [35, 6]. In contrast, the approaches we develop are specialized to particular extensions, and thus have specific advantages over an axiomatic approach in practice.

The idea of using inconsistent (partial) models that guide the learning of new facts is not new. For example, the CDCL algorithm of modern SAT solvers learns clauses to eliminate inconsistent assignments; branch-and-bound in integer programming learns lemmas to eliminate real solutions; and the decision procedure for the theory of arrays [15] generates expensive array lemmas based on the current model. The MCSAT approach to SMT, as another example, [20] is based entirely on the interplay of models and lemmas that refute them. Although our approach is similar in spirit, our goals are different. All mentioned approaches are targeting concrete theories where saturation with lemmas is complete and the models are used to guide control the saturation. Our approach, on the other hand, targets generic theories where a decision procedure is either not available or incompatible with DPLL(T). The advantage of the presented framework is that reasoning in complex theories can be achieved by relying on existing DPLL(T) technology supported by the majority of existing SMT solvers (solving the base theory, relying on equality reasoning and simplification), and very little additional engineering effort to generate relevant refinement lemmas.

A number of SMT solvers support string reasoning [39, 31, 36, 1]. Techniques for extended string constraints [12, 39, 36] rely on eager reductions to a core language of basic constraints. To our knowledge, no other string solvers leverage context-dependent simplification. Recent lightweight approaches for non-linear arithmetic constraints have been explored in [3, 17]. Current state-of-the-art approaches for bit-vectors rely on eager bit-blasting techniques with approaches.An earlier approach for lazy bit-blasting was proposed by Bruttomesso et. al [16]. A recent approach for bit-vectors uses lazy bit-blasting based on the MCSAT framework is given by Zeljic et al [38].

8 Conclusion and Future Work

We have presented new approaches for handling constraints in the theories of strings, bit-vectors, and non-linear arithmetic. The common thread in each of these approaches is to partition the signatures of these signatures into a basic and extended parts, and treat constraints in the extended signature using context-dependent simplification and model-based refinement techniques. Our evaluation indicates that these techniques are highly

effective for an extended theory of strings and give CVC4 some advantages with the state-of-the-art for non-linear arithmetic. Our preliminary results suggest the approach may be promosing for bit-vectors as well.

We plan use these techniques in part to develop further theory extensions that would be useful to support in SMT solvers. Other extensions of interest worth pursuing include a stratified approach for floating-point constraints, commonly used type conversion functions (e.g. bv_to_int, int_to_str), and transcendental functions.

Acknowledgments We would like to thank Liana Hadarean and Martin Brain for helpful discussion about bit-vectors, and Tim King for his support for arithmetic in CVC4.

References

[1] P. A. Abdulla, M. F. Atig, Y. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman. Norn: An SMT solver for string constraints. In *CAV*, Lecture Notes in Computer Science, pages 462–469. Springer, 2015.

[2] B. Akbarpour and L. C. Paulson. Metitarski: An automatic theorem prover for real-valued special functions. *Journal of Automated Reasoning*, 44(3):175–205, 2010.

[3] J. Avigad, R. Y. Lewis, and C. Roux. A heuristic prover for real inequalities. *J. Autom. Reasoning*, 56(3):367–386, 2016.

[4] T. Ball and J. Daniel. Deconstructing dynamic symbolic execution. In *Proceedings of the 2014 Marktoberdorf Summer School on Dependable Software Systems Engineering*. IOS Press, 2014.

[5] K. Bansal, A. Reynolds, C. W. Barrett, and C. Tinelli. A new decision procedure for finite sets and cardinality constraints in SMT. In *IJCAR*, pages 82–98, 2016.

[6] K. Bansal, A. Reynolds, T. King, C. W. Barrett, and T. Wies. Deciding local theory extensions via e-matching. In *CAV*, pages 87–105, 2015.

[7] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *CAV*, CAV'11, pages 171–177. Springer-Verlag, 2011.

[8] C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015. Available at `www.SMT-LIB.org`.

[9] C. Barrett, P. Fontaine, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[10] C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting on demand in SAT modulo theories. In *Proceedings of LPAR'06*, pages 512–526. Springer, 2006.

[11] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. Satisfiability modulo theories. In A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185, chapter 26, pages 825–885. IOS Press, February 2009.

[12] N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *TACAS*, pages 307–321. Springer-Verlag, 2009.

[13] M. Brain, V. D'Silva, A. Griggio, L. Haller, and D. Kroening. Deciding floating-point logic with abstract conflict driven clause learning. *Formal Methods in System Design*, 2014.

[14] R. Brummayer and A. Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *TACAS*, TACAS '09, pages 174–177, 2009.

[15] R. Brummayer and A. Biere. Lemmas on demand for the extensional theory of arrays. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:165–201, 2009.

[16] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, Z. Hanna, A. Nadel, A. Palti, and R. Sebastiani. A lazy and layered SMT(BV) solver for hard industrial verification problems. In *CAV*, pages 547–560, 2007.

[17] A. Cimatti, A. Griggio, A. Irfan, M. Roveri, and R. Sebastiani. Invariant checking of NRA transition systems via incremental reduction to LRA with EUF. In *TACAS*, 2017.

[18] M. A. Colón, S. Sankaranarayanan, and H. B. Sipma. Linear invariant generation using non-linear constraint solving. In *CAV*, pages 420–432. Springer, 2003.

[19] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, TACAS'08/ETAPS'08, pages 337–340. Springer-Verlag, 2008.

[20] L. De Moura and D. Jovanović. A model-constructing satisfiability calculus. In *VMCAI*, pages 1–12. Springer, 2013.

[21] B. Dutertre. Yices 2.2. In *International Conference on Computer Aided Verification*, pages 737–744. Springer, 2014.

[22] B. Dutertre and L. De Moura. A fast linear-arithmetic solver for DPLL(T). In *CAV*, pages 81–94. Springer Berlin Heidelberg, 2006.

[23] C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. Sat solving for termination analysis with polynomial interpretations. In *SAT*, 2007.

[24] V. Ganesh, M. Minnes, A. Solar-Lezama, and M. Rinard. Word equations with length constraints: What's decidable? In *HVC*, HVC'12, pages 209–226. Springer-Verlag, 2013.

[25] J. Giesl, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, et al. Proving termination of programs automatically with aprove. In *IJCAR*, pages 184–191. Springer, 2014.

[26] L. Hadarean, K. Bansal, D. Jovanovic, C. Barrett, and C. Tinelli. A tale of two solvers: Eager and lazy approaches to bit-vectors. In *CAV*, pages 680–695, 2014.

[27] D. Jovanović. Solving nonlinear integer arithmetic with mcsat. In *VMCAI*, pages 330–346. Springer, 2017.

[28] D. Jovanović and L. De Moura. Solving non-linear arithmetic. In *International Joint Conference on Automated Reasoning*, pages 339–354, 2012.

[29] T. King. *Effective algorithms for the satisfiability of quantifier-free formulas over linear real and integer arithmetic*. PhD thesis, Courant Institute of Mathematical Sciences New York, 2014.

[30] J. Leike and M. Heizmann. Ranking templates for linear loops. In *TACAS*, pages 172–186. Springer, 2014.

[31] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. In *CAV*, 2014.

[32] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: from an abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, Nov. 2006.

[33] A. Reynolds and J. C. Blanchette. A decision procedure for (co)datatypes in SMT solvers. In *CADE*, 2015.

[34] A. Reynolds, M. Woo, C. Barrett, D. Brumley, T. Liang, and C. Tinelli. Scaling up DPLL(T) string solvers using context-dependent simplification. In *CAV*, 2017. (To appear).

[35] V. Sofronie-Stokkermans. Hierarchic reasoning in local theory extensions. In *CADE*, pages 219–234, 2005.

[36] M.-T. Trinh, D.-H. Chu, and J. Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In M. Yung and N. Li, editors, *Proceedings of the 21st ACM Conference on Computer and Communications Security*, 2014.

[37] T. Van Khanh and M. Ogawa. Smt for polynomial constraints on real numbers. *Electronic Notes in Theoretical Computer Science*, 289:27–40, 2012.

[38] A. Zeljic, C. M. Wintersteiger, and P. Rümmer. Deciding bit-vector formulas with mcsat. In *SAT 2016*, pages 249–266, 2016.

[39] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: A z3-based string solver for web application analysis. In *Foundations of Software Engineering*, ESEC/FSE 2013, 2013.