TLP **17** (4): 516–558, 2017. © Cambridge University Press 2017
doi:10.1017/S1471068417000175 First published online 28 June 2017

*Constraint solving for finite model finding in SMT solvers**

ANDREW REYNOLDS and CESARE TINELLI
Department of Computer Science, The University of Iowa, Iowa City, Iowa, USA
(*e-mail*: andrew.j.reynolds@gmail.com, cesare-tinelli@uiowa.edu)

CLARK BARRETT
Department of Computer Science, Stanford University, Stanford, California, USA
(*e-mail*: barrett@cs.stanford.edu)

submitted 2 June 2017; revised 2 June 2017; accepted 2 June 2017

Abstract

Satisfiability modulo theories (SMT) solvers have been used successfully as reasoning engines for automated verification and other applications based on automated reasoning. Current techniques for dealing with quantified formulas in SMT are generally incomplete, forcing SMT solvers to report "unknown" when they fail to prove the unsatisfiability of a formula with quantifiers. This inability to return counter models limits their usefulness in applications that produce queries involving quantified formulas. In this paper, we reduce these limitations by integrating finite model finding techniques based on constraint solving into the architecture used by modern SMT solvers. This approach is made possible by a novel solver for cardinality constraints, as well as techniques for on-demand instantiation of quantified formulas. Experiments show that our approach is competitive with the state of the art in SMT, and orthogonal to approaches in automated theorem proving.

KEYWORDS : Satisfiability modulo theories, finite model finding

1 Introduction

Satisfiability modulo theories (SMT) is a subfield of automated reasoning concerned with the problem of determining the satisfiability of formulas in some first-order theory T, where T is usually a combination of several sub-theories. SMT techniques and solvers have been used successfully in recent years to support a variety of formal methods for hardware and software development, including automated verification. They are especially effective for tasks that can be reduced to proving the unsatisfiability of *quantifier-free* formulas in certain theories, such as theories of linear arithmetic, algebraic datatypes, bit vectors, arrays, strings and so on, for which it is possible to build specialized constraint solvers. A number of applications, however, require reasoners that can prove the unsatisfiability of *quantified* formulas

* The work of the first two authors was partially funded by a grant from Intel Corporation.

in those theories. In verification applications, for instance, quantified formulas are necessary to express properties of systems with an unbounded number of processes, or properties involving a number of memory locations. In general, the need for quantified formulas arises naturally when dealing with function or predicate symbols that do not belong to the signature of an SMT solver's built-in theories.

The few SMT solvers that can currently reason about quantified formulas are based on incomplete methods and often report "unknown" when they fail, after some predetermined amount of effort, to prove a quantified formula unsatisfiable. For many client applications, however, it is very useful to know if the failure is due to the fact that the input formula is indeed satisfiable, especially if the solver can also return some representation of the formula's model. Current SMT solvers are able to produce models of satisfiable quantified formulas only in fairly restricted cases (Ge and de Moura 2009), which limits their scope and usefulness.

We reduce these limitations with a novel approach for model finding in SMT. Since, by the undecidability of first-order logic, there are no automated methods for finding arbitrary models, we focus on *finite* models, which can be represented symbolically and enumerated. More precisely, because SMT solvers work with sorted logics with both built-in and *free* (uninterpreted) sorts, we focus on finding models that interpret the free sorts as finite domains. As with traditional finite model finders for first-order logic, the main idea is simply to check universally quantified formulas exhaustively over candidate models with increasingly large domains for the free sorts, until an actual model is found. Our approach differs from previous ones in that it does not rely on the explicit introduction of *domain constants* for the free sorts, as done by MACE-style model finders (Claessen and Sörensson 2003), and in that we are able to reason modulo more theories than just the theory of equality, unlike SEM-style model finders (Zhang and Zhang 1995). Moreover, and crucially for our goals, the approach is fully integrated into the general architecture underlying most SMT solvers.

While limited to SMT formulas with quantifiers ranging only over free sorts or built-in finite sorts (such as, for instance, bit vector sorts or enumeration sorts), our approach is still quite useful. Formulas with such quantifiers occur often, for instance, in verification applications; moreover, when they are satisfiable they usually have small finite models.

We present our model finding method in the context of an abstract framework that models a large class of SMT solvers supporting multiple theories and quantified formulas (Krstić and Goel 2007). We incorporate in this framework an efficient mechanism for deciding the satisfiability of a set of ground SMT formulas under *finite cardinality* constraints (FCC) for the free sorts. This is used first to find a candidate model, a model $\mathcal{M}$ of a heuristically generated finite set of ground consequences of the input formula φ. To check that $\mathcal{M}$ satisfies φ as well, the model finder then verifies, in a complete way, that *all* ground consequences of φ over the universe of $\mathcal{M}$ are satisfied by $\mathcal{M}$. When this check fails, the model finder looks for a new candidate model, possibly under extended cardinality bounds for the free sorts. The practical effectiveness of this approach relies on two crucial components: (*i*) a method for constructing and representing candidate models efficiently and (*ii*) a model-based quantifier instantiation heuristic that avoids the explicit generation and checking of all the ground instances of the input formula. The two are strictly related since the instantiation heuristic takes advantage of the way candidate models are represented to identify, and ignore, entire sets of instances that do not need to be considered.

The paper is organized as follows. After discussing preliminaries in Section 3, we present the framework used by SMT solvers in Section 4. We then present a high-level overview of our approach for finite model finding in Section 5, followed by details in Sections 6–8 This includes, in particular, the strategy used by the solver for finding small candidate finite models and the algorithm for checking the satisfiability of quantified formulas against these candidate models. Section 9 describes an experimental evaluation of our implementation of these techniques in the SMT solver cvc4 on several sets of benchmarks.

This paper builds on material from previous conference papers (Reynolds *et al.* 2013a; Reynolds *et al.* 2013b), as well as the PhD dissertation by the first author Reynolds (2013).

2 Related work

Most traditional finite model finders for quantified formulas are based on a reduction to a decidable logic, propositional logic or some decidable fragments of first-order logic, where the reduction introduces finite upper bounds on the cardinalities of the atomic types. This technique was pioneered by McCune in the Mace tool (McCune 1994), and is often referred to as MACE-style model finding. These techniques were later implemented in the tool Paradox (Claessen and Sörensson 2003), which incorporated successful techniques for symmetry breaking. Other conceptually similar tools include FM-Darwin (Baumgartner *et al.* 2009), which handles first-order logic with equality, the Alloy Analyzer with its backend Kodkod (Torlak and Jackson 2007) which handles first-order relational logic, and Nitpick (Blanchette and Nipkow 2010) which handles higher order logic. Recently, a MACE-style finite model finding approach was also implemented in the Vampire theorem prover (Reger *et al.* 2016).

A different approach to model finding, pioneered by the SEM model finder (Zhang and Zhang 1995), does not encode the input problem into propositional logic. Instead, it uses built-in support for equality together with constraint propagation techniques similar to those used in modern constraint solvers. In this respect, our approach is more similar to SEM-style model finding than it is to MACE-style model finding.

Our approach uses on-demand quantifier instantiation to check the satisfiability of universally quantified formulas. Other instantiation-based approaches have been developed, both in the automated theorem proving community (Korovin 2008) and in SMT. For the latter, instantiation-based techniques are most typically used in an incomplete way for finding proofs of unsatisfiability (Detlefs *et al.* 2003; de Moura and Bjørner 2007; Ge *et al.* 2009). Other techniques establish the satisfiability of quantified formulas, either by using model-based techniques (Ge and de Moura

https://doi.org/10.1017/S1471068417000175 Published online by Cambridge University Press

2009), or by reasoning in local theories where only a finite set of instances is required for completeness (Ihlemann *et al.* 2008).

3 Preliminaries

We work in the context of many-sorted first-order logic with equality. A (many-sorted) *signature* Σ consists of a set of sort symbols and a set of *(sorted) function symbols*, $f : S_1 \times \cdots \times S_n \rightarrow S$, where $n \geqslant 0$ and $S_1, \ldots, S_n, S$ are sorts in Σ. When n is 0, f is also called a *constant symbol*. We use the binary predicate $\approx$ to denote equality. We assume that Σ always includes a Boolean sort Bool and constants true and false of that sort—allowing us to encode all other predicate symbols as function symbols of return sort Bool. For such symbols, we may write, e.g., $P(t_1, \ldots, t_n)$ as shorthand for the equality $P(t_1, \ldots, t_n) \approx \mathsf{true}$. A signature Σ_0 is a *subsignature* of a signature Σ, and Σ is a *supersignature* of Σ_0, if every sort and function symbol of Σ_0 is also in Σ.

Given a signature Σ, a Σ-term is either a (sorted) variable x with sort from Σ, or an expression of the form $f(t_1, \ldots, t_n)$, where f is a function from Σ, and $t_1, \ldots, t_n$ are Σ-terms. A term t is a *well-sorted* term of sort S if t is a variable having sort S, or t is of the form $f(t_1, \ldots, t_n)$ where f is of sort $S_1 \times \cdots \times S_n \rightarrow S$, and $t_1, \ldots, t_n$ are well-sorted terms of sorts $S_1, \ldots, S_n$, respectively. An *atomic* Σ*-formula* is an equality $t_1 \approx t_2$ where t_1 and t_2 are well-sorted Σ-terms of the same sort. A Σ-*literal* is either an atomic Σ-formula p or its negation $\neg p$. We write $s \not\approx t$ as an abbreviation for $\neg s \approx t$. A Σ-*clause* is a disjunction of Σ-literals, e.g., $l_1 \vee \ldots \vee l_n$. A Σ-*formula* is an expression built from atomic Σ-formulas logical connectives such as $\vee$, $\wedge$, and $\neg$, and quantifiers $\forall$ and $\exists$. An occurrence of variable x is *free* in a formula φ if it does not reside within a sub-formula $\forall x\,\psi$ or $\exists x\,\psi$ of φ. We write $FV(\varphi)$ to denote the set of variables that occur free in φ, or the *free variables* of φ. A Σ-*sentence* is a Σ-formula with no free variables. A Σ-term or formula is *ground* if it contains no variables. More generally, by a slight abuse of terminology, we will sometimes call ground any quantifier-free term or formula. Where $\mathbf{x} = (x_1, \ldots, x_n)$ is a tuple of sorted variables, we write $\forall \mathbf{x}\,\varphi$ as an abbreviation for $\forall x_1 \cdots \forall x_n\,\varphi$ and $\exists \mathbf{x}\,\varphi$ as an abbreviation for $\neg\forall\mathbf{x}\,\neg\varphi$. When using this notation, we will implicitly assume that $\mathbf{x}$ is maximal—for example, we assume that $\forall x_1\,\forall x_2\,\varphi$ is instead written as $\forall x_1 x_2\,\varphi$.

A substitution σ is a mapping from variables to terms, applied in postfix form, such that $x\sigma$ and x have the same sort for every variable x and the set $\mathscr{D}om(\sigma) := \{x \mid x\sigma \neq x\}$, the *domain* of σ, is finite. We say σ is a *most general unifier* of terms t_1 and t_2 if σ is a substitution with minimal domain such that $t_1\sigma = t_2\sigma$.

A Σ-*interpretation* $\mathscr{I}$ maps each sort S in Σ to a non-empty set $S^{\mathscr{I}}$, the *domain* of S in $\mathscr{I}$; it maps each variable x of sort S to an element $x^{\mathscr{I}}$ of $S^{\mathscr{I}}$ and each function symbol $f : S_1 \times \cdots \times S_n \rightarrow S \in \Sigma$ to a total function $f^{\mathscr{I}} : S_1^{\mathscr{I}} \times \cdots \times S_n^{\mathscr{I}} \rightarrow S^{\mathscr{I}}$. If Σ_0 is a sub-signature of Σ, the Σ_0-*reduct* of $\mathscr{I}$ is the Σ_0-interpretation $\mathscr{I}_0$ that interprets the symbols of Σ_0 exactly as $\mathscr{I}$ does. The evaluation of a term $f(t_1, \ldots, t_n)$ in $\mathscr{I}$, denoted $\mathscr{I}[\![t]\!]$, is defined recursively as (*i*) $\mathscr{I}[\![x]\!] = x^{\mathscr{I}}$; (*ii*) $\mathscr{I}[\![f(t_1, \ldots, t_n)]\!] = f^{\mathscr{I}}(\mathscr{I}[\![t_1]\!], \ldots \mathscr{I}[\![t_n]\!])$. For a Σ-interpretation $\mathscr{I}$, a variable x of sort S, and an element u of $S^{\mathscr{I}}$, we write $\mathscr{I}[x \rightarrow u]$ to denote a Σ-interpretation that interprets x as u, and is

https://doi.org/10.1017/S1471068417000175 Published online by Cambridge University Press

otherwise identical to $\mathcal{I}$. The usual satisfiability relation $\models$ between Σ-interpretations and Σ-formulas, written $\models$, is defined as follows[1]:

- $\mathcal{I} \models t_1 \approx t_2$ iff $\mathcal{I}[\![t_1]\!] = \mathcal{I}[\![t_2]\!]$.
- $\mathcal{I} \models \varphi \wedge \psi$ iff $\mathcal{I} \models \varphi$ and $\mathcal{I} \models \psi$.
- $\mathcal{I} \models \varphi \vee \psi$ iff $\mathcal{I} \models \varphi$ or $\mathcal{I} \models \psi$.
- $\mathcal{I} \models \neg\varphi$ iff $\mathcal{I} \not\models \varphi$.
- $\mathcal{I} \models \forall x\, \varphi$ iff $\mathcal{I}[x \to v] \models \varphi$ for all $v \in S^{\mathcal{I}}$ where S is the sort of x.

A Σ-interpretation $\mathcal{M}$ *satisfies* (or *is a model of*) a Σ-formula φ if $\mathcal{M} \models \varphi$; $\mathcal{M}$ *satisfies* (or *is a model of*) a set of Σ-formulas if it satisfies all of them. A Σ-formula or set of Σ-formulas is *satisfiable* if it has a model and is *unsatisfiable* otherwise. We consider only interpretations that interpret Bool as a binary set and true and false as distinct elements of that set. We write $\perp$ to abbreviate the unsatisfiable formula $\mathsf{false} \approx \mathsf{true}$. A set Γ of formulas *propositionally entails* a formula φ, written $\Gamma \models_p \varphi$, if the set $\Gamma \cup \{\neg\varphi\}$ is unsatisfiable when considering all atomic formulas in it as propositional (Boolean) variables.

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, closed under variable reassignment (that is, for all $\mathcal{I} \in \mathbf{I}$, every Σ-interpretation that differs from $\mathcal{I}$ only in how it interprets variables is also in $\mathbf{I}$). The *union* of two theories $T_1 = (\Sigma_1, \mathbf{I}_1)$ and $T_2 = (\Sigma_2, \mathbf{I}_2)$, when it exists, is the theory $T_1 \cup T_2 = (\Sigma, \mathbf{I})$ where Σ is the smallest super-signature of Σ_1 and Σ_2 and $\mathbf{I}$ is the set of all Σ-interpretations whose Σ_i-reduct is in $\mathbf{I}_i$ for $i = 1, 2$. This definition extends to more than two theories as expected.

Given a theory $T = (\Sigma, \mathbf{I})$, a Σ-formula φ is *satisfiable modulo* T, or T-*satisfiable*, if and only if there is a model of T that satisfies φ. A set Γ of Σ-formulas T-*entails* a Σ-formula φ, written $\Gamma \models_T \varphi$, if every model of T that satisfies all formulas in Γ satisfies φ as well. The formula φ is T-*valid* if it is T-entailed by the empty set—equivalently, if it is satisfied by every model of T. Two sets Γ_1 and Γ_2 of Σ-formulas are *equisatisfiable in* T if for every model of T that satisfies one there is a model of T that satisfies the other, and the two models differ at most on the way they interpret the free variables not shared by Γ_1 and Γ_2.

4 The DPLL($T_1, \ldots, T_m$) framework

Most SMT solvers have a basic architecture that combines in a principled way a propositional satisfiability solver, the *SAT engine*, with a number of *theory* solvers, specialized constraint solvers for sets of literals over a specific theory. A general framework, called DPLL(T), to describe at an abstract but formal level the working of such SMT solvers and the interaction of their main components was originally developed by Nieuwenhuis *et al.* (2006). The framework, parameterized by a background theory T, describes entire families of procedures to determine the

[1] Cases for additional constructs such as $\Rightarrow$, $\Leftrightarrow$ and $\exists$ can be defined as usual by reduction to the cases below.

https://doi.org/10.1017/S1471068417000175 Published online by Cambridge University Press

T-satisfiability of a ground set of input clauses. We present here a variant of it, introduced by Krstić and Goel (2007), where T is not a monolithic theory but is instead the union of a number of separate sub-theories, each with its own theory solver.

4.1 The theory T

For the rest of the paper, we will consider a Σ-theory $T = T_1 \cup \cdots \cup T_m$ where each T_i is a theory with signature Σ_i. One of these theories, say T_e with $e \in \{1,\ldots,m\}$, may be the theory of equality—over the symbols Σ_e. This theory, whose set of models consists of all Σ_e-interpretations, is also known as the theory of equality with uninterpreted functions (EUF). As a consequence, we will refer to the sort and function symbols of T that occur only in Σ_e as *uninterpreted* and to the other symbols of T as *interpreted*. To stress that we treat the component theories of T individually, we will refer to our variant of DPLL(T) as DPLL($T_1,\ldots,T_m$).

For convenience and without loss of generality, we assume that if a signature from $\{\Sigma_1,\ldots,\Sigma_m\}$ shares a sort symbol with another signature then it shares it also with all the signatures in the set. Finally, we impose the (true) restriction that the signatures $\Sigma_1,\ldots,\Sigma_m$ share no function symbols at all except for true and false. This restriction is currently imposed by all SMT solvers that support multiple theories as it enables the modular combination of theory solvers for the individual theories.

4.2 Transition system

The DPLL($T_1,\ldots,T_m$) framework for theory T defines a state transition system for each ground Σ-formula φ_0 whose T-satisfiability one is interested in. Intuitively, the initial state of the system corresponds to a CNF encoding of φ_0. Under the right conditions on T, all of the executions of the system starting from such a state end in a distinguished fail state if and only if φ_0 is not T-satisfiable.

States System states are all triples of the form $\langle M, F, C\rangle$ where

- M, the current *assignment*, is a sequence of literals and *decision points* $\bullet$,
- F is a set of ground clauses derived from φ_0, and
- C is either the distinguished value no or a clause, which we will refer to as a *conflict clause.*

Each assignment M can be factored uniquely into the subsequence concatenation $M_0 \bullet M_1 \bullet \cdots \bullet M_n$, where no M_i contains decision points. For $i = 0,\ldots,n$, we call M_i the *decision level* i of M and denote with $M^{[i]}$ the subsequence $M_0 \bullet \cdots \bullet M_i$. When convenient, we will treat M as the set of its literals. The formulas in F have a particular *purified form* that can be assumed with no loss of generality since any formula can be efficiently converted into that form while preserving equisatisfiability in T: each element of F is a ground clause, and each atom occurring in F is *pure*, that is, has signature Σ_i for some $i \in \{1,\ldots,m\}$. By the way, assignments are constructed, their atoms too are always pure.

Initial states have the form $\langle \emptyset, F_0, \mathsf{no}\rangle$ where F_0 is an input set of clauses to be checked for T-satisfiability. The expected final states are states of the form

https://doi.org/10.1017/S1471068417000175 Published online by Cambridge University Press

$$\textsf{Propagate}_i \quad \frac{l_1,\ldots,l_n \in M \quad l_1,\ldots,l_n \models_i l \quad l \in \mathrm{Lit}_F \cup \mathrm{Int}_M \quad l, \bar{l} \notin M}{M := M\ l}$$

$$\textsf{Decide} \quad \frac{l \in \mathrm{Lit}_F \cup \mathrm{Int}_M \quad l, \bar{l} \notin M}{M := M \bullet l} \qquad \textsf{Conflict}_i \quad \frac{C = \textsf{no} \quad l_1,\ldots,l_n \in M \quad l_1,\ldots,l_n \models_i \bot}{C := \bar{l}_1 \vee \cdots \vee \bar{l}_n}$$

$$\textsf{Explain}_i \quad \frac{C = l \vee D \quad \bar{l}_1,\ldots,\bar{l}_n \models_i \bar{l} \quad \bar{l}_1,\ldots,\bar{l}_n \prec_M \bar{l}}{C := l_1 \vee \cdots \vee l_n \vee D}$$

$$\textsf{Learn}_i \quad \frac{\emptyset \models_i l_1 \vee \cdots \vee l_n \quad l_1,\ldots,l_n \in \mathrm{Lit}_M|_i \cup \mathrm{Int}_M \cup L_i}{F := F \cup \{l_1 \vee \cdots \vee l_n\}} \qquad \textsf{Learn}_0 \quad \frac{C \neq \textsf{no} \quad \bullet \in M}{F := F \cup \{C\}}$$

$$\textsf{Backjump} \quad \frac{C = l_1 \vee \cdots \vee l_n \vee l \quad \textsf{lev}\ \bar{l}_1,\ldots,\textsf{lev}\ \bar{l}_n \leqslant i < \textsf{lev}\ \bar{l}}{C := \textsf{no} \quad M := M^{[i]}\ l} \qquad \textsf{Fail} \quad \frac{C \neq \textsf{no} \quad \bullet \notin M}{C := \bot}$$

Fig. 1. DPLL($T_1,\ldots,T_m$) rules

$\langle M, F, \bot\rangle$, when F_0 is not T-satisfiable; or else $\langle M, F, \textsf{no}\rangle$ with M satisfiable in T, F equisatisfiable with F_0 in T, and $M \models_{\mathrm{p}} F$.

If M is T-satisfiable and $M \models_{\mathrm{p}} F$, we call M a *satisfying assignment* for F.

Transition rules. The possible behaviors of the system are defined by a set of non-deterministic state transition rules, specifying a set of successor states for each current state. The rules are provided in Figure 1 in *guarded assignment form* (Krstić and Goel 2007). A rule applies to a state s if all of its premises hold for s. In the rules, M, F, and C, respectively, denote the assignment, formula set, and conflict clause component of the current state. The conclusion describes how each component is changed, if at all.

We write $\bar{l}$ to denote the complement of literal l and write $l \prec_M l'$ to indicate that l occurs before l' in M. The function $\textsf{lev}$ maps each literal of M to the (unique) decision level at which l occurs in M. The set Lit_F (resp., Lit_M) consists of all literals in F (resp., all literals in M) and their complements. For $i = 1,\ldots,m$, the set $\mathrm{Lit}_M|_i$ consists of the Σ_i-literals of Lit_M. Int_M is the set of all *interface literals* of M: the equalities and disequalities between *shared variables* where the set of shared variables is

$$\{x \mid x \text{ is a variable in both } \mathrm{Lit}_M|_i \text{ and } \mathrm{Lit}_M|_j \text{for some } 1 \leqslant i < j \leqslant m\}\ .$$

The index i in the rules ranges from 0 to m for **Propagate**$_i$, **Conflict**$_i$, and **Explain**$_i$, and from 1 to m for **Learn**$_i$. In all rules, $\models_i$ abbreviates $\models_{T_i}$ when $i > 0$. In **Propagate**$_0$, $l_1,\ldots,l_n \models_0 l$ simply means that $\bar{l}_1 \vee \cdots \vee \bar{l}_n \vee l \in F$. Similarly, in **Conflict**$_0$, $l_1,\ldots,l_n \models_0 \bot$ means that $\bar{l}_1 \vee \cdots \vee \bar{l}_n \in F$; in **Explain**$_0$, $\bar{l}_1,\ldots,\bar{l}_n \models_0 \bar{l}$ means that $l_1 \vee \cdots \vee l_n \vee \bar{l} \in F$.

The rules **Decide**, **Propagate**$_0$, **Explain**$_0$, **Conflict**$_0$, **Learn**, and **Backjump** model the behavior of the SAT engine, which treats atoms as Boolean variables. The rules **Conflict**$_0$ and **Explain**$_0$ model the conflict discovery and analysis mechanism used by CDCL SAT solvers. All the other rules model the interaction between the SAT engine and the individual theory solvers in the overall SMT solver.

Generally speaking, the system uses the SAT engine to construct the assignment M as if the problem were propositional, but it periodically asks the sub-solvers for each theory T_i to check if the set of Σ_i-literals in M is T_i-unsatisfiable, or entails in T_i some yet undetermined literal from $\mathrm{Lit}_F \cup \mathrm{Int}_M$. In the first case, the sub-solver returns an *explanation* of the unsatisfiability as a conflict clause, which is modeled by **Conflict**$_i$ with $i \in \{1, \dots, m\}$. The propagation of entailed theory literals and the extension of the conflict analysis mechanism to them is modeled by the rules **Propagate**$_i$ and **Explain**$_i$.

The inclusion of the interface literals Int_M in **Decide** and **Propagate**$_i$ achieves the effect of the Nelson–Oppen combination method (Tinelli and Harandi 1996; Bruttomesso *et al.* 2009). Under the right conditions on the component theories, the two rules allow the overall system to determine the T-satisfiability of the input formula by doing only local reasoning in the individual component theories and exchanging information between their corresponding solvers just through (dis)equalities between interface variables.

The rule **Learn**$_i$ with $i > 0$ is needed to model theory solvers following the splitting-on-demand paradigm (Barrett *et al.* 2006). When asked about the satisfiability of their constraints, these solvers may instead return a *splitting lemma*, a T_i-valid formula that encodes an additional guess that needs to be made about the literals in M before the solver can determine their satisfiability. The set L_i in the rule is a finite set consisting of literals, not present in the input set F_0, which may be generated by such solvers.

4.3 System executions and correctness

An *execution* of a transition system modeled as above is a (possibly infinite) sequence $s_0, s_1, \dots$ of states such that s_0 is an initial state and for all $i \geqslant 0$, s_{i+1} can be generated from s_i by the application of one of the transition rules. A system state is *reachable* if it occurs in some execution; it is *irreducible* if no transition rules besides **Learn**$_i$, apply to it. An *exhausted execution* is a finite execution whose last state is irreducible. A *complete execution* is either an exhausted execution or an infinite execution. An application of **Learn**$_i$, with $i \geqslant 0$, is *redundant* in an execution if the execution contains a previous application of **Learn**$_i$ with the same premise.

Adapting results from Barrett *et al.* (2006), Krstić and Goel (2007), and Nieuwenhuis *et al.* (2006), it can be shown that every execution starting with a state $\langle \emptyset, F_0, \mathsf{no} \rangle$ and ending in $\langle M, F, C \rangle$ satisfies the following invariants:

(1) M contains only pure literals and no repetitions.

(2) $F \models_T C$ and $M \models_{\mathrm{p}} \neg C$ when $C \neq \mathsf{no}$.

(3) F_0 and F are equisatisfiable in T.

https://doi.org/10.1017/S1471068417000175 Published online by Cambridge University Press

Moreover, the transition system is *terminating*: every execution with no redundant applications of **Learn**$_i$ is finite; and *sound*: for every execution starting with a state $\langle \emptyset, F_0, \mathsf{no} \rangle$ and ending with a state $\langle M, F, \bot \rangle$, the clause set F_0 is T-unsatisfiable. Under suitable assumptions on the sub-theories $T_1, \ldots, T_m$, the system is also *complete*: for every exhausted execution starting with $\langle \emptyset, F_0, \mathsf{no} \rangle$ and ending with $\langle M, F, \mathsf{no} \rangle$, M is satisfiable in T and $M \models_{\mathrm{p}} F_0$. Here, we provide a sketch of the correctness proof for DPLL($T_1, \ldots, T_m$) restricted to a single theory T_1 based on the proof for the original framework (Nieuwenhuis *et al.* 2006).

Theorem 1

Suppose $T = T_1$. With any strategy where all applications of **Learn**$_1$ are not redundant and introduce new literals only from a finite set L_1, DPLL(T_1) is sound, complete, and terminating for all sets F_0 of ground clauses.

Proof: (Sketch)

Soundness. Observe that all reachable states of the form $\langle M, F, C \rangle$ where $C \neq \mathsf{no}$ are such that C is T_1-entailed by F_0. When C is introduced by **Conflict**$_0$, it is a clause from F; when it is introduced by **Conflict**$_1$, it is T_1-valid. When applying **Explain**$_i$, we replace a literal l in C with disjunction of literals $l_1 \vee \cdots \vee l_n$ which is entailed by l either in the theory T_1 (when $i = 1$), or together with F (when $i = 0$). Thus, when a state of the form $\langle M, F, \bot \rangle$ is reachable, we can conclude that $F \models_{T_1} \bot$. Since all clauses in F are T_1-entailed by F_0 by construction, $\bot$ is T_1-entailed by F_0 as well.

Termination. For all reachable states $\langle M, F, C \rangle$, every literal occurring in M, F, or C belongs to the finite set of literals $\mathrm{Lit}_M \cup \mathrm{Int}_M \cup L_1$. As a consequence, there is only a finite number of reachable states. Consider a partial ordering $\succeq$ on assignments M, with the empty assignment as maximal element, such that $(e_1\, M_1) \succeq (e_2\, M_2)$ if either (i) $e_1 = \bullet$ and $e_2 \neq \bullet$ or (ii) $e_1 = e_2$ and $M_1 \succeq M_2$. In addition, consider a partial ordering $\succeq$ on conflict clauses, seeing as sets of literals, such that $C_1 \succeq C_2$ if either C_1 is no, or neither C_1 nor C_2 are no and $C_2 \prec_M^{mul} C_1$, where $\prec_M^{mul}$ is the multiset extension of $\prec_M$. Extend this ordering to states so that $\langle M_1, F_1, C_1 \rangle \succeq \langle M_2, F_2, C_2 \rangle$ if and only if $M_1 \succeq M_2$, or $M_1 = M_2$ and $C_1 \succeq C_2$. One can show that this ordering is well founded. Moreover, applying all rules besides **Learn**$_1$ to a state s results in a state s' where $s \succ s'$. Termination then follows from the fact that **Learn**$_1$ is applicable only a finite number of times.

Completeness. We claim that for every irreducible reachable state $\langle M, F, \mathsf{no} \rangle$, M is a T_1-satisfiable satisfying assignment for F. To see this, consider first that since **Decide** does not apply to the state, M must contain an assignment for all literals in F. Moreover, M is a satisfying assignment for F since **Conflict**$_0$ does not apply. Since **Conflict**$_1$ does not apply, then M must be T_1-satisfiable. Since $F_0 \subseteq F$, we have that M propositionally entails F_0, and thus F_0 is T_1-satisfiable as well. Thus, since our procedure is terminating, it is also complete. □

The soundness and termination arguments in the proof above immediately extend to the multiple theory case of DPLL($T_1, \ldots, T_m$) where $m > 1$. The completeness argument can be extended as well under further model-theoretic assumptions on the component theories (Krstić and Goel 2007; Jovanovic and Barrett 2013).

https://doi.org/10.1017/S1471068417000175 Published online by Cambridge University Press

```
proc check($M, F, C$) $\equiv$
  (Propagate$_0$ | … | Propagate$_n$)$^*$;
  if weak_effort($M, F, C$) = true
    if $l, \bar{l} \notin M$ for some $l \in \mathrm{Lit}_F$
      Decide on $l$
    else if strong_effort($M, F, C$)
      return $\langle M, F, \mathsf{no} \rangle$
  return check_conflict($M, F, C$)

proc check_conflict($M, F, C$) $\equiv$
  if $C \neq$ no
    (Explain$_0$ | … | Explain$_n$)$^*$;
    if $C = \emptyset$
      return $\langle M, F, \bot \rangle$
    else
      Learn; Backjump
  return check($M, F, C$)
```

Fig. 2. A typical strategy check for applying DPLL($T_1, \ldots, T_m$) rules.

4.4 A typical strategy for DPLL($T_1, \ldots, T_m$)

A typical strategy for applying the theory-specific rules of DPLL($T_1, \ldots, T_m$) is outlined in Figure 2 where | denotes alternative choice and * denotes zero or more rule applications. The check procedure involves two sub-procedures weak_effort and strong_effort, which are not shown here and are specific to the theory T. Each of these methods when invoked either applies **Conflict**$_i$ or **Learn**$_i$ for some $1 \leqslant i \leqslant m$ and returns false, or applies no rules and returns true.

Weak effort checks, as denoted by weak_effort, are commonly used to eagerly avoid extensions that are clearly unsatisfiable in one of the theories. On the other hand, *strong effort checks*, as denoted by strong_effort, are required to make progress toward determining the T-satisfiability of the conjunction of literals in M. In particular, unlike weak_effort, we require that strong_effort returns true only when M is T-satisfiable. Generally speaking, weak effort checks typically involve computationally inexpensive reasoning at the cost of incompleteness, whereas strong effort checks are complete but may involve expensive reasoning. The design of a theory solver in the DPLL($T_1, \ldots, T_m$) framework depends largely on how the methods weak_effort and strong_effort are implemented. We will see an example of these functions in Section 6.2.

In more detail, the first sub-procedure check in Figure 2 applies to states $\langle M, F, C \rangle$ where $C =$ no. We first apply the rule **Propagate**$_i$ for sub-theories T_i, possibly multiple times. Afterwards, we apply a weak effort check. If no conflicts or clauses are learned at weak effort, we apply **Decide** on some unassigned literal l from Lit_F, if one exists. Otherwise, our assignment M is complete, and we apply a strong effort check to verify the T-satisfiability of M. If strong_effort(M, F, C) returns true, then M is satisfiable in T, and the method returns the (final) state $\langle M, F, \mathsf{no} \rangle$, indicating that F is satisfiable. In all other cases, we apply check_conflict.

The second sub-procedure check_conflict is applied to states $\langle M, F, C \rangle$ where C may be different from no. In those cases, we perform conflict analysis by repeated applications of **Explain**$_i$. If we reach the fail state, then we know F is unsatisfiable. Otherwise, we add a learned clause via **Learn**, and apply **Backjump** to return to a prefix of M.

We formally state the requirements of weak and strong effort checks for the single theory case in the following proposition which is a consequence of Theorem 1.

https://doi.org/10.1017/S1471068417000175 Published online by Cambridge University Press

Proposition 1
Suppose $T = T_1$. The check method in Figure 2 implements a sound, complete, and terminating strategy for all sets F_0 of ground clauses provided all of the following hold:

(1) In $\mathsf{weak\_effort}$ and $\mathsf{strong\_effort}$, all applications of $\textbf{Learn}_1$ are not redundant and introduce new literals only from a finite set L_1.
(2) $\mathsf{weak\_effort}$ and $\mathsf{strong\_effort}$ return false only when they apply at least one rule.
(3) $\mathsf{strong\_effort}(M, F, C)$ returns true only when the conjunction of the literals in M is T_1-satisfiable.

Proof: (Sketch) The first point ensures that check meets the requirements on applications of $\textbf{Learn}_1$ as given in Theorem 1; the second point ensures that check is a terminating method; and the third point ensures that check generates exhaustive executions. □

5 Finite model finding in SMT

The DPLL($T_1, \ldots, T_m$) framework described in the previous section is limited to quantifier-free formulas. This section outlines an approach for finite model finding for quantified formulas that can be integrated in DPLL($T_1, \ldots, T_m$)-based SMT solvers. Concretely, we consider Σ-formulas in the following language:

$$\phi := t_1 \approx t_2 \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \forall x\, \varphi,$$

where t_1 and t_2 are Σ-terms, and *the sort of x is either an uninterpreted sort or a sort interpreted in every model of T as a finite set of some fixed cardinality*. Examples of the latter include sorts denoting fixed-length bit-vectors or finite (non-recursive) datatypes. Certain integer arithmetic constraints with bounded quantifiers, where the bounds are explicitly provided or can be inferred, can be treated similarly to finite interpreted sorts (Reynolds 2013; Baumgartner *et al.* 2014). Many applications of SMT rely on solving problems that fall into such categories, given a careful encoding of the constraints.

Given an input formula ψ in the grammar above, our approach first performs a purification step, which results in a set F of ground clauses, and a set A of equivalences of the form $a \approx \mathsf{true} \Leftrightarrow \forall \mathbf{x}\, \varphi$, abbreviated as $a \Leftrightarrow \forall \mathbf{x}\, \varphi$, where a is a Boolean variable uniquely associated with the quantified formula $\forall \mathbf{x}\, \varphi$. We will refer to a as the *proxy variable* for $\forall \mathbf{x}\, \varphi$. The set F can be constructed by a standard conversion of ψ to clausal form which, however, treats the quantified sub-formulas of ψ as atoms. After that, each quantified formula $\forall \mathbf{x}\, \varphi$ occurring in a clause of F is replaced with its proxy variable a if it occurs positively in the clause, and with $\varphi\{\mathbf{x} \mapsto \mathbf{k}\}$ otherwise, where $\varphi\{\mathbf{x} \mapsto \mathbf{k}\}$ is the result of substituting each occurrence in φ of a variable x of $\mathbf{x}$ with a fresh variable k (to be treated like a Skolem constant). The process is repeated until F contains no quantifiers. This conversion from ψ to F and A can be done so that ψ and $F \cup A$ are equisatisfiable. We will denote by $\lfloor \psi \rfloor$ the resulting pair (F, A).

https://doi.org/10.1017/S1471068417000175 Published online by Cambridge University Press

```
FM-Solve$_{\mathscr{H}}$ $(F, A)$
Input: A set $F$ of purified $\Sigma$-clauses and a set $A$ of equivalences
Output: sat or unsat

1. Find a satisfying assignment $M$ for $F$. If none is found, return unsat.
2. Construct a $\Sigma$-interpretation $\mathscr{M}$ that satisfies $M$. Let $\mathbf{V}$ be a minimal set of $\Sigma$-terms
   such that, for each uninterpreted sort $S$ of $\Sigma$, every element of $S^{\mathscr{M}}$ is denoted by a
   term in $\mathbf{V}$ (that is, for all $s \in S^{\mathscr{M}}$ there is a $v \in \mathbf{V}$ such that $s = \mathscr{M}[\![v]\!]$).
3. For each $\forall \mathbf{x}\, \varphi$ where $a \Leftrightarrow \forall \mathbf{x}\, \varphi \in A$ and $a \in M$,
   (a) let $I_{\mathbf{x}}$ be the set of substitutions from $\mathbf{x}$ to terms in $\mathbf{V}$ chosen by $\mathscr{H}(\mathscr{M}, \forall \mathbf{x}\, \varphi)$;
   (b) let $(F, A) = (F \cup F', A \cup A')$ where $(F', A') = \lfloor \{\neg a \vee \varphi\sigma \mid \sigma \in I_{\mathbf{x}}\} \rfloor$.
   If each of the sets $I_{\mathbf{x}}$ was empty, return sat, otherwise go to Step 1.
```

Fig. 3. Finite Model Finding Procedure FM-Solve$_{\mathscr{H}}$, parameterized by a quantifier instantiation heuristic $\mathscr{H}$.

Example 1

Consider the formula $\psi = \neg P(b, c) \wedge (Q(b, c) \Leftrightarrow \forall x\, P(b, x))$ where P and Q are uninterpreted predicates and x, b, and c are of some uninterpreted sort S. The purified form $\lfloor \psi \rfloor$ is computed as follows. First, a conversion of ψ to clausal normal form results in the clauses:

$$F_0 := \{\neg P(b, c),\ \neg Q(b, c) \vee \forall x\, P(b, x),\ Q(b, c) \vee \neg \forall x\, P(b, x)\}.$$

We replace the occurrence of $\neg \forall x\, P(b, x)$ in the third clause of F_0 with $\neg P(b, k)$, where k is a fresh variable of sort S, and replace the positive occurrence in the second clause with a fresh proxy variable a of sort Bool. We obtain the quantifier-free set of purified clauses F and equivalences A:

$$\begin{array}{lcl} F & := & \{\neg P(b, c), \neg P(b, k) \vee Q(b, c), a \vee \neg Q(b, c)\}, \\ A & := & \{a \Leftrightarrow \forall x\, P(b, x)\}. \end{array}$$

It is not hard to see that ϕ and $F \cup A$ are equisatisfiable in T.

5.1 Model finding procedure

Figure 3 describes a finite model finding procedure called FM-Solve$_{\mathscr{H}}$ that takes as input a set F and a set A, where $(F, A) = \lfloor \phi \rfloor$ for some Σ-formula ψ, and tries to determine the satisfiability of $F \cup A$ by adding to F instances of the quantified formulas that occur in A. The procedure is parameterized by an instantiation heuristic $\mathscr{H}$ for the quantified formulas.

In Step 1, it looks for a satisfying assignment M for F[2]. This assignment can be found using the DPLL$(T_1, \ldots, T_m)$ procedure from the previous section. If

[2] Recall that a satisfying assignment is a T-satisfiable set of Σ-literals that propositionally entail F.

https://doi.org/10.1017/S1471068417000175 Published online by Cambridge University Press

no satisfying assignment can be found, the procedure terminates with unsat, for "unsatisfiable." Otherwise, in Step 2, it constructs a Σ-interpretation $\mathscr{M}$ that satisfies M. In doing so, however, it considers only Σ-interpretations $\mathscr{M}$ that interpret the uninterpreted sorts of Σ as finite sets. This makes it feasible to actually construct the set $\mathbf{V}$ used in Step 3. In that step, the procedure considers the set of quantified formulas that are *active* in M, that is, those whose proxy variable occurs positively in M. It adds new constraints $F' \cup A'$ to $F \cup A$ based on instances of quantified formulas chosen by the heuristic $\mathscr{H}$, which takes as input a model and a quantified formula. We consider only heuristics that are sound with respect to models: if $\mathscr{H}$ returns no instances for quantified formulas in Step 3, it is because $\mathscr{M}$ satisfies all active quantified formulas in M, and so the procedure terminates with sat, for "satisfiable."

Theorem 2

For all inputs F, A for FM-Solve$_{\mathscr{H}}$, the following hold:

(1) If the method for finding satisfying assignments M for F in Step 1 is sound, then the procedure FM-Solve$_{\mathscr{H}}$ returns unsat only if $F \cup A$ is T-unsatisfiable.

(2) If for all inputs, $\mathscr{H}(\mathscr{M}, \forall \mathbf{x}\, \varphi)$ returns the empty set only if $\mathscr{M} \models \forall \mathbf{x}\, \varphi$, then the procedure FM-Solve$_{\mathscr{H}}$ returns sat only if $F \cup A$ is T-satisfiable.

Proof: To show Point 1, assume the method for finding satisfying assignments M for F in Step 1 is sound. Thus, when the procedure returns unsat, we have that F is T-unsatisfiable. Since the formulas added to F and A in Step 3 preserve satisfiability, we have that our input is T-unsatisfiable as well.

To show Point 2, the procedure returns sat when $\mathscr{H}(\mathscr{M}, \forall \mathbf{x}\, \varphi)$ returns the empty set for all quantified formulas where $a \Leftrightarrow \forall \mathbf{x}\, \varphi \in A$ and $a \in M$. Assume $\mathscr{M} \models \forall \mathbf{x}\, \varphi$ for all such formulas. Then, $F \cup A$ is satisfied by a model $\mathscr{M}'$ where $a^{\mathscr{M}'} = (\forall \mathbf{x}\, \varphi)^{\mathscr{M}}$ for each $a \Leftrightarrow \forall \mathbf{x}\, \varphi \in A$ and $a \notin M$, and where all other symbols are interpreted as in $\mathscr{M}$. Since during all iterations of the procedure $F \cup A$ remains a superset of the original input, we have that the latter is satisfied by $\mathscr{M}'$ as well. □

The following sections will examine in more detail the main ideas behind the three steps of procedure FM-Solve$_{\mathscr{H}}$. In Section 6, we describe techniques for finding satisfying assignments in Step 1. These assignments have models that interpret uninterpreted sorts as sets of minimal cardinality and are used in Step 2. In Section 7, we describe methods for constructing such models. Finally, in Section 8, we describe quantifier instantiation heuristics that can be used to choose sets of substitutions for Step 3.

Although Theorem 2 holds in general for inputs that involve several theories, for simplicity, we restrict ourselves in the following to problems in EUF only, that is, involving only uninterpreted sorts and function symbols. Under these restrictions, we provide arguments for the correctness of the three steps of FM-Solve$_{\mathscr{H}}$ in Theorems 4 and 5, which ensure the correctness of our finite model finding procedure according to Theorem 2.

https://doi.org/10.1017/S1471068417000175 Published online by Cambridge University Press

6 EUF with finite cardinality constraints (FCC)

In this section, we introduce techniques for finding satisfying assignments in Step 1 of procedure $\mathsf{FM\text{-}Solve}_{\mathscr{H}}$ from Figure 3. We will focus on satisfying assignments that have *small* models, that is, models which interpret the uninterpreted sorts of our signature as finite sets of minimal size. To do this, we introduce an extension of the theory EUF with FCC. We describe its signature (Σ_{FCC}) and semantics, give a satisfiabiliy procedure for conjunctions of literals in this theory, and describe how it can be integrated into the DPLL($T_1, \dots, T_m$) architecture. Finally, we discuss a strategy, *fixed-cardinality* $\mathsf{check}_{\mathrm{FCC}}$, which ensures that upper bounds are incrementally established for all uninterpreted sorts.

Defnition 1 (FCC)
Let EUF be the theory of equality and uninterpreted functions over some signature Σ_{EUF}. The theory FCC of EUF *with FCC* is the extension of EUF obtained as follows. The signature Σ_{FCC} of FCC extends Σ_{EUF} with a constant $\mathsf{card}_{S,k}$ of sort Bool for each sort S of Σ_{EUF} and integer $k > 0$. Its models are all Σ_{FCC}-interpretations that satisfy each atomic formula $\mathsf{card}_{S,k}$ exactly when they interpret S as a set of cardinality at most k.

As shown below, the FCC-satisfiability of sets of Σ_{FCC}-literals is a decidable problem. By a reduction from graph (vertex) coloring, one can show that the problem is NP-hard. The main idea of the reduction is to represent the set of k colors as a sort C and represent the vertices of the graph as variables of sort C. An edge between two vertices x and y is encoded as the constraint $x \not\approx y$. The cardinality constraint on C is encoded by $\mathsf{card}_{\mathsf{C},k}$. It is not difficult to see that given a model $\mathscr{M}$ of FCC (which is finitely representable), checking whether $\mathscr{M}$ satisfies a set of Σ_{FCC}-literals can be done in polynomial time. It follows that this satisfiability problem is NP-complete.

We prove its decidability by providing an effective satisfiability procedure. The procedure relies on computing certain congruence closures of sets of constraints, so we start by introducing that notion.

Definition 2 (*Congruence closure*)
Let M be a set of literals, in any signature, and let $\mathbf{T}_M$ be the set of all terms (and sub-terms) occurring in M. The *congruence closure* M^* *of* M is the smallest set of literals such that

(1) $M \subseteq M^* \subseteq \{s \approx t \mid s,t \in \mathbf{T}_M\} \cup \{s \not\approx t \mid s,t \in \mathbf{T}_M\}$;
(2) for all $s,t \in \mathbf{T}_M$, $M^* \models_{\mathrm{EUF}} s \approx t$ iff $s \approx t \in M^*$.

By construction, the relation $\{(s,t) \mid s \approx t \in M^*\}$ induced by M^* is a congruence, and hence an equivalence, relation over $\mathbf{T}_M$. For brevity, we will identify M^* with its induced equivalence relation when convenient. It can be shown (see, e.g., Baader and Nipkow 1998) that (*i*) M^* is computable whenever M is finite, and (*ii*) if M is satisfiable it is satisfied by an interpretation $\mathscr{M}$ that interprets each sort S as the set $\mathbf{V}_S = \{v_1^S, \dots, v_{n_S}^S\}$ consisting of an arbitrary representative v_i^S for each of the n_S equivalence classes of M^* over terms of sort S. We call $\mathscr{M}$ a *normal model* of M. Our procedure will seek to find normal models for given input sets M of literals.

https://doi.org/10.1017/S1471068417000175 Published online by Cambridge University Press

Input: A set M of Σ_{FCC}-literals
Output: sat or unsat

1. If $s \approx t \in M^*$ for some $s \not\approx t \in M \cup \{\mathsf{false} \not\approx \mathsf{true}\}$, return unsat.
2. If M contains no positive cardinality literals, return sat; otherwise, let k be the smallest integer such that $\mathsf{card}_{\mathsf{S},k} \in M$.
3. If $\neg\mathsf{card}_{\mathsf{S},j} \in M$ for some $j \geqslant k$, return unsat.
4. If there are k or fewer equivalence classes in M^*, return sat.
5. If there exists two terms s and t in distinct equivalence classes of M^* such that $M \not\models_{\mathrm{EUF}} s \not\approx t$, run the procedure recursively on $M \cup s \approx t$ and $M \cup s \not\approx t$, returning sat if either of the two subcalls returns sat, and returning unsat otherwise.
6. Return unsat.

Fig. 4. Decision Procedure for FCC.

6.1 Decision procedure

This section presents a decision procedure for the satisfiability of sets of constraints in the theory FCC. For now, we limit ourselves to signatures Σ_{FCC} whose set of sorts consists of a single (uninterpreted) sort S. Figure 4 gives the decision procedure for the satisfiability problem in this case. As input, the procedure takes a set M consisting of cardinality constraint literals for S and equalities and disequalities over ground Σ_{FCC}-terms of sort S, and terminates with sat or unsat.

Lemma 1
The procedure in Figure 4 is sound, complete, and terminating for every set M of Σ_{FCC}-literals.

Proof: *Soundness.* Let us start by observing that splitting the problem based on equalities $s \approx t$, as done in Step 5 of the procedure, is trivially sound since all models of FCC satisfy exactly one of $s \approx t$ and $s \not\approx t$. The procedure answers unsat in one of the following cases:

(1) An equality $s \approx t$ is entailed by M where $s \not\approx t$ is also in $M \cup \{\mathsf{false} \not\approx \mathsf{true}\}$.
(2) Conflicting literals $\mathsf{card}_{\mathsf{S},k}$ and $\neg\mathsf{card}_{\mathsf{S},j}$ are asserted for $j \geqslant k$.
(3) There exist $k+1$ terms (each in a different equivalence class) that are entailed to be mutually disequal by M.

For the first case, it is immediate that M has no models. For conflicts in the second case, no model can be constructed with both at most k and at least $j+1$ elements in the domain of S. For conflicts in the third case, note that if the procedure reaches Step 6, there must be $k+1$ equivalence classes with representatives $t_1, \ldots, t_{k+1}$, say, where $M \models_{\mathrm{EUF}} t_i \not\approx t_j$ for all $1 \leqslant i < j \leqslant k+1$; hence no model can be constructed satisfying $\mathsf{card}_{\mathsf{S},k}$.

Termination. It is easy to see that when the procedure recurses in Step 5, the set of equalities and disequalities in M without the cardinality constraints is satisfiable. Let C be a set collecting the equivalence classes of M^* and let $[t]_M$ denote the equivalence class of a term t. We argue that the splitting on the equality of s and t

done at Step 5 decreases the size of the set

$$E_M := \{ ([u]_M, [v]_M) \in C \times C \mid [u]_M \neq [v]_M,\ M \not\models_{\text{EUF}} u \not\approx v \} \tag{6.1}$$

in other words, it decreases the number of equivalence classes that are pairwise not entailed to be disequal. In either branch of the split on $s \approx t$, no equivalence classes are created (although two existing ones are possibly merged), and $([s]_M, [t]_M)$ is no longer an element of E_M in the recursive call. When E_M becomes empty, the procedure is guaranteed to terminate, since either more than k equivalence classes are entailed to be distinct, in which case the procedure answers unsat, or there are at most k equivalence classes, in which case the procedure answers sat.

Completeness. The procedure answers sat when the congruence closure M^* contains no equality whose negation occurs in M, and either there is no positive cardinality literal in M, or M^* has at most k equivalence classes where k is the smallest integer such that $\mathsf{card}_{\mathsf{S},k} \in M$. In either case, we can construct a model where S is interpreted as a set of size j, with $j \leqslant k$ for all $\mathsf{card}_{\mathsf{S},k} \in M$ and $j \geqslant k$ for all $\neg\mathsf{card}_{\mathsf{S},k} \in M$. If j is greater than the number of equivalence classes in M^*, arbitrary new elements can be added to the domain of S without affecting the satisfiability of the equalities and disequalities in M. □

An immediate consequence of this lemma is that constraint satisfiability in FCC is decidable.

Proposition 2
The FCC-satisfiability of sets of Σ_{FCC}-literals is decidable.

The completeness argument in Lemma 1 also suggests a constructive proof of the following result.

Proposition 3
Every satisfiable set of Σ_{FCC}-literals has a finite model.

We point out that in the absence of cardinality constraints, the decision procedure in Figure 4 reduces to the standard congruence closure procedure used to decide the satisfiability of constraints in EUF. SMT solvers supporting EUF have theory solvers that essentially implement that procedure.

6.2 *Integration into DPLL*$(T_1, \ldots, T_m)$

Our decision procedure for FCC can be integrated into the DPLL$(T_1, \ldots, T_m)$ framework by capitalizing on the existence of a theory solver for EUF (T_e). We effectively extend such a solver modularly with facilities to reason about cardinality constraints as well. Since FCC is an extension of EUF, we now replace the latter with the former in the framework and make $T_e = $ FCC. Recall the strategy outlined in Figure 2 of Section 4.4. In the following, we detail how the methods weak_effort and strong_effort of this strategy are implemented for FCC.

For simplicity, we maintain the restriction for now that FCC contains a single uninterpreted sort S. Also, when convenient, we identify equivalence classes of terms with their representative terms.

https://doi.org/10.1017/S1471068417000175 Published online by Cambridge University Press

```
proc weak_effort_FCC $(M, F, C)$ $\equiv$
  if $l_1, \ldots, l_n \models_{\mathrm{EUF}} \bot$ for some $l_1, \ldots, l_n \in M$
    Apply Conflict$_e$ with $C := \bar{l}_1 \vee \cdots \vee \bar{l}_n$, return false
  else if $\mathsf{card}_{\mathsf{S},k}, \neg\mathsf{card}_{\mathsf{S},j} \in M$ for $j > k$,
    Apply Conflict$_e$ with $C := \neg\mathsf{card}_{\mathsf{S},k} \vee \mathsf{card}_{\mathsf{S},j}$, return false
  else if $\mathsf{card}_{\mathsf{S},k} \in M$ and $M \models_{\mathrm{EUF}} \mathsf{distinct}(t_1, \ldots, t_{k+1})$
    Apply Learn$_e$ to $\neg\mathsf{card}_{\mathsf{S},k} \vee \neg\mathsf{distinct}(t_1, \ldots, t_{k+1})$, return false
  else
    return true
```

Fig. 5. Weak effort check for FCC.

6.2.1 Weak effort check

At weak effort, we recognize conflicting states of three different forms, outlined in Figure 5. First, if we are unable to construct a congruence closure for M that is consistent with the disequalities from M, we identify a subset $\{l_1, \ldots, l_n\}$ of M that is EUF-unsatisfiable and apply **Conflict**$_e$ to it. Second, if M contains the conflicting cardinality constraints $\mathsf{card}_{\mathsf{S},k} \in M$ and $\neg\mathsf{card}_{\mathsf{S},j}$ with $j > k$, we construct the conflict clause $\neg\mathsf{card}_{\mathsf{S},k} \vee \mathsf{card}_{\mathsf{S},j}$. Third, we may recognize cases when M contains a literal of the form $\mathsf{card}_{\mathsf{S},k}$ while its other literals entails that $k+1$ terms $t_1, \ldots, t_{k+1}$ are pairwise disequal. In this case, we use **Learn**$_e$ to add the lemma $\neg\mathsf{card}_{\mathsf{S},k} \vee \neg\mathsf{distinct}(t_1, \ldots, t_{k+1})$ to the current set F of clauses, where $\mathsf{distinct}(t_1, \ldots, t_{k+1})$ is shorthand for the conjunction of disequalities stating that the terms $t_1, \ldots, t_{k+1}$ are pairwise distinct[3]. We will refer to a lemma of this form as a *clique lemma*. We assume that this instance of **Learn**$_e$ is applied only when the resultant clause does not occur in F. In practice, this can be achieved either by maintaining a cache of learned clauses or by ensuring **Propagate**$_e$ is applied to completion between each call. We apply **Learn**$_e$ because the constraint $\neg\mathsf{distinct}(t_1, \ldots, t_{k+1})$ may contain literals not belonging to F. We could alternatively apply **Conflict**$_e$ to construct a conflict clause of form $\bar{l}_1 \vee \ldots \vee \bar{l}_n \vee \neg\mathsf{card}_{\mathsf{S},k}$, where $\{l_1, \ldots, l_n\}$ is a subset of M that entails $\mathsf{distinct}(t_1, \ldots, t_{k+1})$. However, we have found that in practice this is inefficient, as many different sets of literals can be found for essentially the same conflict.

Generating clique lemmas. For the purposes of discovering and learning clique lemmas, we incrementally construct and maintain on the side a *disequality graph* D for S, whose vertices correspond to the equivalence classes of terms of sort S induced by the congruence closure of M, and whose edges represent disequalities in M between terms in different equivalence classes. In this representation, a sufficient condition for discovering a conflict reduces to finding a $(k + 1)$-clique in D. Now, even just checking for the presence of a $(k + 1)$-clique in a n-vertex graph is too expensive in general—as this is an NP-complete problem (Garey *et al.* 1974). For this reason, the weak effort check of our procedure uses an incomplete check for potential cliques. This is done by partitioning the vertices of the graph into suitable

[3] Note that $\neg\mathsf{card}_{\mathsf{S},k} \vee \neg\mathsf{distinct}(t_1, \ldots, t_{k+1})$ is a valid formula of FCC.

subsets that we call *regions*. After defining regions formally, we explain below how we exploit them to discover clique-related conflicts efficiently in practice.

Definition 3 (*k-Region*)
Let $D = (V, E)$ be an undirected graph and let R be a subset of V. For all vertices $v \in R$, let $\mathsf{ext}(v)$ be the number of edges between v and vertices not in R. We say R is a *k-region* of D if for all $0 < i \leqslant k$, the size of the set $\{v \mid v \in R, \mathsf{ext}(v) \geqslant i\}$ is smaller than $k-i$. A *k-regionalization* $\mathscr{R}_D$ of D is a partition of V into k-regions. We will refer to it simply as a *regionalization* when k is understood or not important.

Regionalizations are useful for us because they facilitate the discovery of cliques.

Lemma 2
If $\mathscr{R}_D$ is a k-regionalization of a graph D and D contains a k-clique C, then all the vertices of C reside in the same region of $\mathscr{R}_D$.

Proof: If $k \leqslant 1$, the statement is trivial. Otherwise, assume by contradiction D contains k-clique $C = C_1 \cup C_2$ for non-empty C_1, C_2 where, for some region R of $\mathscr{R}_D$, $v \in R$ for all $v \in C_1$ and $v \notin R$ for all $v \in C_2$. Say $|C_2| = i$, and thus $|C_1| = k-i$. Since C is a k-clique, $\mathsf{ext}(v)$ must be at least i for all $v \in C_1$, contradicting the assumption that R is a region. □

Notice that any graph $D = (V, E)$ has a trivial regionalization, with just one region which contains all vertices in V.

Example 2
Consider the constraints $\{c_1 \not\approx c_2, c_2 \not\approx c_3, c_3 \not\approx c_4\}$, all over sort S, and the partition $\{\{c_1, c_2\}, \{c_3, c_4\}\}$. This partition is a 3-regionalization in the disequality graph induced by this set, because a 3-clique can span two regions only if it contains two vertices with interregional edges, and this partition only has one such edge. Adding the disequality $c_2 \not\approx c_4$ or $c_1 \not\approx c_4$ breaks the regionalization invariant.

Let us examine how to maintain a k-regionalization in an (initially empty) *evolving graph* D, a data structure supporting the dynamic allocation of vertices and edges, as well as the merging of vertices. In our framework, where D's vertices correspond to equivalence classes of terms and edges to disequalities between them, these operations are triggered by operations performed on the data structure that stores the congruence closure M^* of the current assignment M. In particular, a vertex v_e is added to D when a new equivalence class e is created, which happens whenever a new term is added to M; an edge between the vertices v_1 and v_2 corresponding to equivalence classes e_1 and e_2 is added to D when the disequation $t_1 \not\approx t_2$ is added to M, for some term t_1 in e_1 and t_2 in e_2; and two vertices are merged, in a single vertex that inherits their edges, when their corresponding equivalence classes are merged during the computation of M^*.

We maintain at all times a $(k+1)$-regionalization $\mathscr{R}_D$ of the graph D, where k is the smallest integer such as $\mathsf{card}_{\mathsf{S},k} \in M$[4]. As the graph D is modified, it may be necessary to merge certain regions of the current within $\mathscr{R}_D$ to ensure the invariant

[4] Recall that $\mathsf{card}_{\mathsf{S},k}$ states that sort S has at most k elements.

https://doi.org/10.1017/S1471068417000175 Published online by Cambridge University Press

```
proc fix_region($R, \mathscr{R}_D$) $\equiv$
  if $R$ is not a $k$-region
    choose some $R' \in \mathscr{R}_D$, where $R' \neq R$
    $\mathscr{R} := \mathscr{R} \setminus \{R, R'\} \cup \{R \cup R'\}$
    fix_region($\{R \cup R'\}, \mathscr{R}_D$)
```

Fig. 6. The fix_region procedure. Ensures $R \in \mathscr{R}_D$ is a k-region by merging it with another $R' \in \mathscr{R}_D$, and repeating this process recursively.

in Definition 3 holds. The procedure fix_region from Figure 6 ensures that a set R within $\mathscr{R}_D$ is a k-region by merging it as needed with another set R' in $\mathscr{R}_D$, and repeating this process recursively until R becomes a k-region. As a heuristic, we choose the R' with the highest density of interregional edges to R.

Assuming we have a regionalization $\mathscr{R}_D$ for graph D, here is how we construct a regionalization $\mathscr{R}_{D'}$ for graph D' resulting from an addition to M. In the following, $\mathscr{R}(v)$ denotes the region in a regionalization $\mathscr{R}$ that contains the vertex v.

Adding Vertices: When a vertex v is added to D, $\mathscr{R}_{D'}$ is the result of adding the singleton region $\{v\}$ to $\mathscr{R}_D$.

Adding Edges: When we add an edge (v_1, v_2) to D, we have that $\mathscr{R}_{D'} = \mathscr{R}_D$ is still a partition of V. However, $\mathscr{R}_D(v_1)$ or $\mathscr{R}_D(v_2)$ may not be regions of D'. We apply the procedure fix_region first to $(\mathscr{R}_D(v_1), \mathscr{R}_{D'})$ and then to $(\mathscr{R}_D(v_2), \mathscr{R}_{D'})$ to ensure that $\mathscr{R}_{D'}$ is a regionalization.

Merging Vertices: When a vertex v_1 is merged with another vertex v_2 in D, we have that D' is a quotient graph of D, that is, D' contains a new vertex, call it u, connected to all vertices that are connected to either v_1 or v_2 in D. If $\mathscr{R}_D(v_1)$ is equal to $\mathscr{R}_D(v_2)$, let R be $(\mathscr{R}_D(v_1) \cup \{u\}) \setminus \{v_1, v_2\}$. Then $\mathscr{R}_{D'}$ is equal to $(\mathscr{R}_D \cup R) \setminus \{\mathscr{R}_D(v_1)\}$. To ensure $\mathscr{R}_{D'}$ is a regionalization, we apply fix_region to $(R, \mathscr{R}_{D'})$. If $\mathscr{R}_D(v_1)$ is not equal to $\mathscr{R}_D(v_2)$, let $\{v_i, v_j\} = \{v_1, v_2\}$, $R_i = (\mathscr{R}_D(v_i) \cup \{u\}) \setminus \{v_i\}$, and $R_j = \mathscr{R}_D(v_j) \setminus \{v_j\}$. Then, $\mathscr{R}_{D'}$ is equal to $(\mathscr{R}_D \cup \{R_i, R_j\}) \setminus \{\mathscr{R}_D(v_1), \mathscr{R}_D(v_2)\}$. We apply fix_region to $(R_i, \mathscr{R}_{D'})$ and subsequently to $(R_j, \mathscr{R}_{D'})$.

Additionally, when $\mathsf{card}_{\mathsf{S},k'}$ is asserted for $k' < k$, we discard the $(k+1)$-regionalization and rebuild a $(k'+1)$-regionalization.

Given a $(k+1)$-regionalization $\mathscr{R}_D$ of D, we will call each region in $\mathscr{R}_D$ with at least $k+1$ vertices a *large region*, and all others *small regions*. For the purposes of efficiently discovering $(k+1)$-cliques during weak effort checks, we maintain a *watched set* of $k+1$ vertices for each large region R in $\mathscr{R}_D$, which we will write as $w(R)$. This set is incrementally updated when vertices are added or removed from regions, and when regions are combined. If there exists a large region R in $\mathscr{R}_D$ where each vertex in $w(R)$ is connected, then we add the clique lemma $\neg\mathsf{card}_{\mathsf{S},k} \vee \neg\mathsf{distinct}(t_1, \ldots, t_{k+1})$ to F using the rule $\textbf{Learn}_e$, where $w(R) = \{t_1, \ldots, t_{k+1}\}$.

6.2.2 *Strong effort check*

Recall from Section 4.4 that a strong effort check must determine that the current set of constraints is consistent, or otherwise report a conflict or lemma. The strong effort

```
**proc** strong_effort_FCC $(M, F, C)$ $\equiv$
  let $k$ be the smallest integer such that $\mathsf{card}_{\mathsf{S},k} \in M$
  let $t_1, \ldots, t_n$ be the equivalence class representatives of sort $\mathsf{S}$ in $M^*$
  **if** $n > k$
    choose $1 \leqslant i < j \leqslant n$ such that $M \not\models_{\mathrm{EUF}} t_i \not\approx t_j$
    apply $\mathbf{Learn}_e$ to $t_i \approx t_j \vee t_i \not\approx t_j$
    return false
  **else**
    return true
```

Fig. 7. Strong effort check for FCC.

check of the FCC solver is given in Figure 7. If $\mathsf{card}_{\mathsf{S},k} \in M$ for some (minimal) k, and there are more than k equivalence class of sort S in the congruence closure of M, then we choose two equivalence class representatives t_i and t_j and apply $\mathbf{Learn}_e$ to add the splitting lemma $(t_i \approx t_j \vee t_i \not\approx t_j)$ to F. In practice, we also insist that future applications of **Decide** on the atom $t_i \approx t_j$ should be invoked with positive polarity. If the number of equivalence classes is less than or equal to k, then the procedure returns true, indicating that M is FCC satisfiable.

The choice of t_i and t_j is guided by the watched set of vertices within regions. In particular, if there is a large region R in $\mathscr{R}_D$, we know that $w(R)$ does not form a clique. We choose t_i, t_j to be two vertices from $w(R)$ that are not connected in D. Otherwise, if there are no large regions in $\mathscr{R}_D$ and there are more than k vertices in D, then there must be at least two small regions. We select two regions R_i and R_j based on a heuristic (namely, the maximum density of interregional edges), combine them into a new region $R_i \cup R_j$, apply fix_region to $R_i \cup R_j$, and repeat.

We illustrate the operation of the FCC solver with a couple of examples.

Example 3

Consider the constraints $\{a \approx f(b), b \approx f(c), a \not\approx b, b \not\approx c, \mathsf{card}_{\mathsf{S},2}\}$ where all terms are over the single sort S. First, the FCC solver computes the congruence $\{\{a, f(b)\}, \{b, f(c)\}, \{c\}\}$. Using a, b, c as the representatives, the solver builds the disequality graph with edges $\{(a,b),(b,c)\}$. Since $\mathsf{card}_{\mathsf{S},2}$ limits the size of S to at most 2, the solver generates the lemma $a \approx c \vee a \not\approx c$. Adding the constraint $a \approx c$ produces no conflicts and allows the FCC solver to answer "satisfiable."

Example 4

Consider the constraints $\{c_1 \approx c, c_4 \approx c, c_1 \not\approx c_2, c_2 \not\approx c_3, c_3 \not\approx c_4, \mathsf{card}_{\mathsf{S},2}\}$ with all constants of sort S. The corresponding disequality graph for these constraints contains a clique of size 3. By discovering that clique, the FCC solver can conclude that it is impossible to shrink the model to two elements, and hence reports a clique lemma of the form $\neg\mathsf{distinct}(c_1, c_2, c_3) \vee \neg\mathsf{card}_{\mathsf{S},2}$.

Because of congruence constraints, guesses on merge lemmas may sometimes lead to inconsistencies when constructing the congruence closure, unless we compute and propagate all entailed disequalities—which is usually not done, for efficiency. This is demonstrated in the following example.

Example 5
Consider the constraints $\{c_3 \approx f(c_1),\ c_4 \approx f(c_2),\ c_3 \not\approx c_4,\ \mathsf{card}_{\mathsf{S},2}\}$ where all the terms have sort S. Unless the EUF sub-solver propagates the entailed literal $c_1 \not\approx c_2$, the FCC solver will construct the disequality graph $(V, E) = (\{c_1, c_2, c_3, c_4\}, \{(c_3, c_4)\})$ for S. Say we decide to apply **Learn**$_e$ on $(c_1 \approx c_2 \vee c_1 \not\approx c_2)$, and then the literal $c_1 \approx c_2$ is added to our set of constraints. The subset $\{c_3 \approx f(c_1),\ c_4 \approx f(c_2), c_3 \not\approx c_4,\ c_1 \approx c_2\}$ will then be found unsatisfiable by congruence closure. In contrast, adding the equalities $c_1 \approx c_3$ and $c_2 \approx c_4$ to our set will produce a model of the required cardinality.

We now state the correctness of our FCC procedure as integrated in the DPLL($T_1, \ldots, T_m$) framework. In the following, we let $\mathsf{check}_{\mathrm{FCC}}$ denote the strategy that applies the rules of DPLL(T_{FCC}) according to Figure 2, with the weak and strong effort checks described in this section.

Theorem 3
$\mathsf{check}_{\mathrm{FCC}}$ is a sound, complete, and terminating strategy for every set of ground clauses F_0.

Proof: Notice that the weak and strong effort methods in this section legally apply DPLL($T_1, \ldots, T_m$) rules, that is, they apply **Conflict**$_e$ only to clauses whose negated literals imply a contradiction and **Learn**$_e$ to clauses that hold in all models. We follow the three requirements for weak and strong effort checks as described in Proposition 1.

To show the first point, the only literals introduced by applications of **Learn**$_e$ (call them L_{FCC}) are equalities and disequalities between terms occurring in F_0. Clearly, L_{FCC} is finite. To show the second point, the weak and strong effort methods in this section false only when they apply at least one rule. To show the third point, $\mathsf{strong\_effort\_FCC}$ (M, F, C) returns true only when the congruence closure of M contains k or fewer equivalence classes for all $\mathsf{card}_{\mathsf{S},k} \in M$. In such states, we are guaranteed that M is satisfiable in FCC. □

6.3 Establishing finite cardinalities

We have now shown that a theory solver for FCC can be integrated into the DPLL($T_1, \ldots, T_m$) architecture with support for eager conflict detection through the use of weak effort checks. In this section, we show an approach that makes use of this solver for answering the following problem: *given an input F, find the smallest integer $n > 0$ such that $F \wedge \mathsf{card}_{\mathsf{S},n}$ is satisfiable.*

A straightforward scheme for solving this problem is the following. First, use the solver to determine if $F \wedge \mathsf{card}_{\mathsf{S},1}$ is satisfiable, and answer satisfiable if so. If this is unsatisfiable, use the solver to determine if $F \wedge \mathsf{card}_{\mathsf{S},2}$ is satisfiable, and so on. Due to Proposition 3, this process is guaranteed to terminate when F is satisfiable. A clear disadvantage of this scheme is that, in the absence of conflict analysis, it diverges when F is unsatisfiable. This section describes an alternative approach that overcomes this limitation. At a high level, our approach modifies the

https://doi.org/10.1017/S1471068417000175 Published online by Cambridge University Press

```
proc weak_effort_fc_FCC $(M, F, C)$ $\equiv$
  Let $k$ be the least $\mathbb{N}$ s.t. $k \geqslant n$ and $\neg\mathsf{card}_{\Sigma,k} \notin M$
  if fix($\mathsf{card}_{\Sigma,k}, M, F, C$) = false
    return false
  For each $\mathsf{S}_i \in \Sigma$, let $k_i$ be the least $\mathbb{N}$ s.t. $\neg\mathsf{card}_{\mathsf{S}_i,k_i} \notin M$
  if fix($\mathsf{card}_{\mathsf{S}_i,k_i}, M, F, C$) = false for a minimal $i$
    return false
  if $k_1 + \ldots + k_n > k$
    Apply Conflict$_e$ to $C := (\vee_{i=1}^{n}\mathsf{card}_{\mathsf{S}_i,k_i-1} \vee \neg\mathsf{card}_{\Sigma,k})$
    return false
  return weak_effort_FCC $(M, F, C)$

proc fix($a, M, F, C$) $\equiv$
  if $a \notin \mathrm{Lit}_F$
    Apply Learn$_e$ to $(a \vee \neg a)$
    return false
  else if $a \notin M$
    Apply Decide to $a$
    return false
  else
    return true
```

Fig. 8. A version of the weak effort check procedure of the FCC solver that fixes the cardinality of uninterpreted sorts $\{\mathsf{S}_1, \ldots, \mathsf{S}_n\}$ in signature Σ according to a fair strategy.

weak effort check of the FCC solver by introducing splits on cardinality constraints $(\mathsf{card}_{\mathsf{S},k} \vee \neg\mathsf{card}_{\mathsf{S},k})$, and deciding upon literals of the $\mathsf{card}_{\mathsf{S},k}$ for the minimal feasible k. Before formally defining this approach, we discuss a generalization that is applicable to signatures with multiple uninterpreted sorts.

6.3.1 Extension to multiple sorts

Consider the case when our signature Σ contains multiple sorts $\mathsf{S}_1, \ldots, \mathsf{S}_n$. Given a set of input clauses F, we wish to determine that either F is unsatisfiable, or find a tuple $(k_1, \ldots, k_n)$ such that $F \wedge \mathsf{card}_{\mathsf{S}_1,k_1} \wedge \ldots \wedge \mathsf{card}_{\mathsf{S}_n,k_n}$ is satisfiable. To find such a tuple, a challenge is to devise a strategy that is *fair*. As an illustrative example, consider the formula $(c \not\approx d \vee \varphi)$, where c and d are constants of sort S_1, and the formula φ does not have a model where S_2 is interpreted as a finite set[5]. Clearly, this formula has a model where the cardinality of sorts S_1 and S_2 are 2 and 1, respectively. However, in the absence of a fair strategy, a naive approach could search for models of size $(1,1)$, $(1,2)$, $(1,3)$, and so on, ad infinitum.

To devise a strategy for finite model finding that is fair in the presence of multiple sorts, we extend the signature Σ of FCC to include *signature cardinality constraints* $\mathsf{card}_{\Sigma,k}$, constants of sort Bool for each integer $k > 0$. Let Σ be a signature containing uninterpreted sorts $\mathsf{S}_1, \ldots, \mathsf{S}_n$. Let $\mathscr{I}$ be a Σ-interpretation that interprets sort $\mathsf{S}_i \in \Sigma$ as a set of size k_i for $1 \leqslant i \leqslant n$. Then, $\mathscr{I}$ satisfies $\mathsf{card}_{\Sigma,k}$ if and only if $k_1 + \ldots + k_n \leqslant k$.

Figure 8 gives an extension of the weak effort check of the FCC solver that introduces cardinality constraints for the purposes of finding small models. In detail, we first find the minimal natural number k such that the literal $\neg\mathsf{card}_{\Sigma,k}$ does not occur in M. Using the sub-routine fix, if the atom $\mathsf{card}_{\Sigma,k}$ does not occur in F, we apply **Learn**$_e$ to add $(\mathsf{card}_{\Sigma,k} \vee \neg\mathsf{card}_{\Sigma,k})$ to F. If it does occur in F, we apply **Decide** to $\mathsf{card}_{\Sigma,k}$. We then do the same for each of the uninterpreted sorts $\mathsf{S}_1, \ldots, \mathsf{S}_n$ in our signature. If these steps do not apply a rule, then M contains the literals $\mathsf{card}_{\Sigma,k}$ and $\neg\mathsf{card}_{\mathsf{S}_i,\ell}$ for each $1 \leqslant \ell < k_i$, $i = 1, \ldots, n$. We then check if $\mathsf{card}_{\Sigma,k}$ is in conflict with the negatively asserted cardinality constraints. In particular, $k_1 + \ldots + k_n > k$,

[5] Observe that φ must contain universal quantifiers for this to be the case.

we return a conflict of the form $(\mathsf{card}_{\mathsf{S}_1,k_1-1} \vee \ldots \vee \mathsf{card}_{\mathsf{S}_n,k_n-1} \vee \neg\mathsf{card}_{\Sigma,k})$, where we write $\mathsf{card}_{\mathsf{S}_i,k_i-1}$ to denote a cardinality constraint when $k_i > 1$ and $\perp$ if $k_i = 1$. Otherwise, we apply the original weak effort check of the FCC solver from Figure 5.

Let *fixed-cardinality* $\mathsf{check}_{\mathrm{FCC}}$ be the strategy that applies the rules of DPLL(T_{FCC}) according to Figure 2 where the weak effort check is the one from Figure 8, and the strong effort check is the one from Figure 7. This strategy maintains the following invariant.

Proposition 4
Given a signature Σ containing uninterpreted sorts $\mathsf{S}_1, \ldots, \mathsf{S}_n$, for each execution of fixed-cardinality $\mathsf{check}_{\mathrm{FCC}}$ ending in $\langle M, F, C \rangle$, either M contains no decision points, or M is of the form $N \bullet \mathsf{card}_{\Sigma,k}\ M_0\ (\bullet\mathsf{card}_{\mathsf{S}_1,k_1} M_1) \cdots\ (\bullet\mathsf{card}_{\mathsf{S}_m,k_m}\ M_m)\ N'$, for some m, $0 \leqslant m \leqslant n$, where N,M_0, …, M_m contain no decision points, N' contains no decision points if $m < n$, $\neg\mathsf{card}_{\Sigma,j} \prec_M \mathsf{card}_{\Sigma,k}$ for each $n \leqslant j < k$, and $\neg\mathsf{card}_{\mathsf{S}_i,j} \prec_M \mathsf{card}_{\mathsf{S}_i,k_i}$ for each $1 \leqslant i \leqslant m$, $1 \leqslant j < k_i$.

In other words, using the strategy fixed-cardinality $\mathsf{check}_{\mathrm{FCC}}$, minimal positive cardinality literals are the first decision literals in satisfying assignments. This invariant follows directly from definition of the method given in Figure 8.

Theorem 4
Fixed-cardinality $\mathsf{check}_{\mathrm{FCC}}$ is a sound, complete, and terminating strategy for every set of ground clauses F.

Proof: Assume our signature Σ contains uninterpreted sorts $\mathsf{S}_1, \ldots, \mathsf{S}_n$. Note that the weak effort check method in Figure 8 extends our original weak effort check while additionally applying only legal applications of DPLL($T_1, \ldots, T_m$) rules, noting we apply **Learn**$_e$ to tautologies of the form $(a \vee \neg a)$, **Conflict**$_e$ to sets of literals that are collectively inconsistent according to our extension of FCC, and **Decide** to literals whose atom does not occur in M. To show this strategy is sound, complete, and terminating, we again follow the three requirements for weak and strong effort checks as given in Proposition 1.

To show the first point, we must show that the set L_{FCC} of literals introduced by applications of **Learn**$_e$ is finite. For each $1 \leqslant i \leqslant n$, let k_i be smallest integer greater than the number of terms of sort S_i in F, and such that the literal $\neg\mathsf{card}_{\mathsf{S}_i,k_i}$ does not occur in F. Let k be the smallest integer greater than $k_1 + \ldots + k_n$, and such that the literal $\neg\mathsf{card}_{\Sigma,k}$ does not occur in F. We claim that the set of literals introduced by applications of **Learn**$_e$, call them L^{fc}_{FCC}, are a subset of the set of all equalities and disequalities between terms from F, the literals of the form $(\neg)\mathsf{card}_{\mathsf{S}_i,j}$ where $1 \leqslant j < k_i$ for each sort S_i, and the literals $(\neg)\mathsf{card}_{\Sigma,j}$ where $n \leqslant j < k$. First, only equalities and disequalities between terms from F are introduced by applications of **Learn**$_e$ for the same reason as in Theorem 3. Second, assume by contradiction that a literal $(\neg)\mathsf{card}_{\mathsf{S}_i,k_i}$ is introduced by an application of **Learn**$_e$. Then, it must be the case that an execution of fixed-cardinality $\mathsf{check}_{\mathrm{FCC}}$ results in a state where $\neg\mathsf{card}_{\mathsf{S}_i,k_i-1} \in M$. For this to be the case, $\neg\mathsf{card}_{\mathsf{S}_i,k_i-1}$ must be added to M by **Propagate**$_e$ or **Backjump**. In either case, there must exist a set of literals $l_1, \ldots, l_n$ from F such that $l_1, \ldots, l_n \models_{\mathrm{FCC}} \neg\mathsf{card}_{\mathsf{S}_i,k_i-1}$. By our selection of k_i, this

https://doi.org/10.1017/S1471068417000175 Published online by Cambridge University Press

is a contradiction since there must be at least k_i terms of sort S_i in F for this to be the case. Third, for similar reasons, by our selection of k, a literal $(\neg)\mathsf{card}_{\Sigma,k}$ cannot be introduced by an application of $\textbf{Learn}_e$ unless there exists an execution of fixed-cardinality $\mathsf{check}_{\mathrm{FCC}}$ resulting in a state where $\neg\mathsf{card}_{\mathsf{S}_i,j} \in M$ for some S_i where $j \geqslant k_i$. This cannot be the case for the reasons mentioned above.

To show the second point, weak effort check method in Figure 8 returns false only when it applies a rule.

To show the third point, the strong effort check of fixed-cardinality $\mathsf{check}_{\mathrm{FCC}}$ is the same as the strong effort check in $\mathsf{check}_{\mathrm{FCC}}$ and thus this holds for the same reason as in the proof of Theorem 3. □

Combining the results of Theorem 4 and Proposition 4, given as input a set of ground Σ-clauses F, fixed-cardinality $\mathsf{check}_{\mathrm{FCC}}$ will terminate either in (*i*) a fail state, establishing that F is unsatisfiable, or (*ii*) a state $\langle M, F, \mathsf{no}\rangle$ where M contains $\mathsf{card}_{\mathsf{S}_i,k_i}$ for each uninterpreted sort S_i in Σ, establishing that F is satisfied by a (finite) model.

For the remainder of the paper, we assume that Step 1 of our finite model finding procedure in Figure 3 uses fixed-cardinality $\mathsf{check}_{\mathrm{FCC}}$ for finding satisfying assignments M.

7 Constructing candidate models

We now focus our attention to Step 2 of procedure $\mathsf{FM\text{-}Solve}_{\mathscr{H}}$ from Figure 3, which attempts to constructs models $\mathscr{M}$ of satisfying assignments M for the input clause set F. We refer to $\mathscr{M}$ as a *candidate model*. Note that the assignment M computed by the procedure may contain occurrences of proxy variables a for quantified formulas $\forall \mathbf{x}\, \varphi$ with the variables in $\mathbf{x}$ ranging over uninterpreted or *finite* sorts. Recall that those formulas are stored in the input set A in equivalences of the form $a \Leftrightarrow \forall \mathbf{x}\, \varphi$. The goal of the procedure is to construct $\mathscr{M}$ so that it satisfies not just M but also all its active quantified formulas (those whose proxy variable a occurs positively in M). The reason is that such a model witnesses the T-satisfiability of $F \cup A$.

To discuss the model construction, we focus on the variables and the uninterpreted sorts and function symbols of Σ, since the interpretation of the other sorts and function symbols is fixed by the theory. We construct a candidate model $\mathscr{M}$ by associating each uninterpreted sort S with a finite set $\mathbf{V}_S$ of *domain elements* (i.e., $S^{\mathscr{M}} = \mathbf{V}_S$). Contrary to other model finding approaches, which use fresh symbols as domain elements, we use the equivalence classes of M^* or, rather, representative terms for these classes. All interpreted sorts are interpreted in $\mathscr{M}$ as usual. We extend M^* to another T-satisfiable set, call it M^*_c, such that the representative of each equivalence class of interpreted sort S_i in M^*_c is a value from $S_i^{\mathscr{M}}$. Such an extension is always possible since M is T-satisfiable.

We then associate each uninterpreted function f of sort $S_1 \times \ldots \times S_n \rightarrow S$ to a function $f^{\mathscr{M}}$ from $S_1^{\mathscr{M}} \times \cdots \times S_n^{\mathscr{M}}$ to $S^{\mathscr{M}}$. We construct this function based on the literals in M that contain f. For instance, if M contains $f(c) \approx b$, then $f^{\mathscr{M}}$ is defined so that it maps the interpretation of c to the interpretation of b. Using those equalities typically produces only a partial definition for f. To complete it, one can

https://doi.org/10.1017/S1471068417000175 Published online by Cambridge University Press

use arbitrary output values for the missing input tuples. We describe choices for doing so in the following.

Concretely, we represent candidate Σ-models with the following data structure.

Definition 4 (*Defining map*)
Let $f : S_1 \times \cdots \times S_n \to S$ be an uninterpreted function symbol of Σ and let $y_1, \ldots, y_n$ be distinct fresh variables of respective sort $S_1, \ldots, S_n$. A *defining map for* f is a finite set Δ_f of well-sorted (directed) equations of the form $f(t_1, \ldots, t_n) \approx v$ with $v \in S^{\mathscr{M}}$ and $t_i \in \{y_i\} \cup S_i^{\mathscr{M}}$ for $i = 1, \ldots, n$, satisfying the following requirements:

(1) If $s_1 \approx v_1, s_2 \approx v_2 \in \Delta_f$ with $s_1 \neq s_2$ and s_1 and s_2 have an most general unifier σ, then

(a) σ is non-empty, and
(b) $s_1\sigma \approx v \in \Delta_f$ for some v.

(2) $f(y_1, \ldots, y_n) \approx v \in \Delta_f$ for some v.

A Σ-*map* is a set $\Delta = \bigcup_{f \in \Sigma^{\mathrm{f}}} \Delta_f$ where each Δ_f is a defining map for f.

For the rest of this section, we will use letters $y, y_1, y_2, \ldots$ to denote variables and $c, c_1, c_2, \ldots$ to denote constant symbols.

Example 6
The set $\{f(c_1, y_2) \approx c_2, f(y_1, c_2) \approx c_1, f(c_1, c_2) \approx c_3, f(y_1, y_2) \approx c_3\}$ is a defining map for f. Notice that $f(c_1, y_2)$ and $f(y_1, c_2)$ have most general unifier $\{y_1 \mapsto c_1, y_2 \mapsto c_2\}$. As required in point 1, this most general unifier is non-empty and an equality of the form $f(c_1, c_2) \approx v$ also occurs in this set.

By construction of Δ, every *flat term*, a Σ-term $t = f(v_1, \ldots, v_n)$ has exactly one *most specific generalization* s among the left-hand sides of the equalities in Δ_f, where s is a generalization of t if $t = s\sigma$ for some substitution σ, and s is more specific than s' if s' is a generalization of s. The existence of this generalization is guaranteed by Point 2 in the definition above; its uniqueness by Point 1. The *value of* t *in* Δ is the value v in the (unique) equality $s \approx v \in \Delta_f$. Thus, a Σ-map Δ represents a normal model $\mathscr{M}$ where each uninterpreted sort S is interpreted as the term set $\mathbf{V}_S$ and each uninterpreted function symbol $f : S_1 \times \cdots \times S_n \to S$ is interpreted as the function $f^{\mathscr{M}}$ mapping every $(v_1, \ldots, v_n) \in S_1^{\mathscr{M}} \times \cdots \times S_n^{\mathscr{M}}$ to the value of $f(v_1, \ldots, v_n)$ in Δ[6].

7.1 Model construction procedure

We now describe a procedure for constructing Σ-maps from satisfying assignments. In particular, we describe a parameterized method for completing the partial definitions (of uninterpreted functions) induced by an assignment M.

Let M be an assignment. Recall that if M is T-satisfiable, it is satisfied by a normal model, that is, a model that interprets each uninterpreted sort S as the set

[6] More precisely, a Σ-map represents a family of normal models which differ only over the variables and the interpreted symbols of Σ.

https://doi.org/10.1017/S1471068417000175 Published online by Cambridge University Press

Input: A satisfying assignment M and a set $U \subseteq \mathbf{T}_M$
Output: A Σ-map Δ

1. For each uninterpreted function symbol $f : S_1 \times \cdots \times S_n \to S$ of Σ,

(a) Let $D_1 = \{f(\mathscr{V}_M(t_1), \dots, \mathscr{V}_M(t_n)) \approx \mathscr{V}_M(t) \mid t \in \mathbf{T}_M,\ t = f(t_1, \dots, t_n)\}$

(b) Let $D_2 = \{f(t_1, \dots, t_n)^\forall \approx \mathscr{V}_M(t) \mid t \in U,\ t = f(t_1, \dots, t_n)\}$

(c) Let $\Delta_f = D_1 \cup D_2$ and let $\{t_i \approx v_i\}_{0 \leqslant i \leqslant m}$ be an arbitrary enumeration of Δ_f. For all $t_i \approx v_i$, $t_j \approx v_j$ with $i < j$ that are unifiable with mgu σ, if $t_i\sigma$ does not already occur as a left-hand side in Δ_f, add $t_i\sigma \approx v_i$ to Δ_f.

(d) If $f(y_1, \dots, y_n)$ does not occur as a left-hand side in Δ_f, add $f(y_1, \dots, y_n) \approx v$ for some arbitrary value $v \in S^{\mathscr{M}}$.

2. Let $\Delta = \bigcup_{f \in \Sigma} \Delta_f$ ■

Fig. 9. Model construction procedure.

$\mathbf{V}_S$ consisting of a representative term for each equivalence class of (the extension of) M's congruence closure M_c^*. For each term t, we write $\mathscr{V}_M(t)$ to denote the representative of t's equivalence class in M_c^*.

For every uninterpreted function symbol $f : S_1 \times \cdots \times S_n \to S$ in Σ, we fix n distinct fresh variables $y_1, \dots, y_n$ of respective sort $S_1 \dots, S_n$. To each uninterpreted sort S, we associate a distinguished ground Σ-term e^S, which we will write as e when S is understood. This ground term guides the selection of default values of the interpretation of uninterpreted function symbols in our model construction procedure, based on the following operation. For a ground Σ-term $f(t_1, \dots, t_n)$, we denote by $f(t_1, \dots, t_n)^\forall$ the term $f(u_1, \dots, u_n)$ where $u_i = y_i$ if $t_i = e$, and $u_i = \mathscr{V}_M(t_i)$ otherwise, for $i = 1, \dots, n$.

The non-deterministic procedure described in Figure 9 constructs a Σ-map from M and a subset U of the set of terms $\mathbf{T}_M$ occurring in M. The subset U determines which terms will be used as the basis for default values of function interpretations. For example, let $\mathscr{M}$ be a normal model induced by the defining map constructed by the procedure in Figure 9 for some U. If $f(e, e) \in U$, then the default value of f in $\mathscr{M}$ is the value of $f(e, e)$ in $\mathscr{M}$. In our implementation of the procedure, we choose the set U to be the entire set $\mathbf{T}_M$, although other choices are possible.

Example 7

Consider an assignment M with the following constraints:

$$\{c_1 \approx f(c_2, e),\ c_3 \approx f(c_4, c_6),\ c_3 \approx f(e, c_4),\ c_6 \approx f(c_2, c_5),\ c_2 \approx c_5,\ c_4 \approx f(e, e)\}$$

where all terms are of uninterpreted sort S. The equivalence classes in the congruence closure of M are

$$\{c_1, f(c_2, e)\},\ \{c_2, c_5\},\ \{c_3, f(c_4, c_6), f(e, c_4)\},\ \{c_4, f(e, e)\}, \{c_6, f(c_2, c_5)\}, \{e\}\ .$$

https://doi.org/10.1017/S1471068417000175 Published online by Cambridge University Press

Let $\mathbf{V}_S = \{c_1, c_2, c_3, c_4, c_6, e\}$ and $U = \{f(c_2, e), f(e, c_4), f(e, e)\}$. Following the procedure to construct the defining map Δ_f, we let

$$\begin{aligned} D_1 &= \{f(c_2, e) \approx c_1, f(c_4, c_6) \approx c_3, f(c_2, c_2) \approx c_6, f(e, e) \approx c_4\} \\ D_2 &= \{f(c_2, y_2) \approx c_1, f(y_1, c_4) \approx c_3, f(y_1, y_2) \approx c_4\} \\ \Delta_f &= D_1 \cup D_2 \end{aligned}$$

Since $f(c_2, y_2)$ and $f(y_1, c_4)$ are unifiable with $\sigma = \{y_1 \mapsto c_2, y_2 \mapsto c_4\}$, and $f(c_2, c_4)$ is not in Δ_f, we add the equality $f(c_2, c_4) \approx c_1$ (alternatively, $f(c_2, c_4) \approx c_3$) to Δ_f. Finally, since $f(y_1, y_2)$ is already in Δ_f, this gives us the set

$$\begin{aligned} \Delta_f = \{ & f(c_2, e) \approx c_1, f(c_4, c_6) \approx c_3, f(c_2, c_2) \approx c_6, f(e, e) \approx c_4, \\ & f(c_2, y_2) \approx c_1, f(y_1, c_4) \approx c_3, f(y_1, y_2) \approx c_4, f(c_2, c_4) \approx c_1\} \end{aligned}$$

which is a complete definition for f. Notice that a different selection of U would have led to a different construction for Δ_f. Let $\mathcal{M}$ be the normal model induced by a Δ containing Δ_f. We have that, for instance, $\mathcal{M}[\![f(c_2, c_3)]\!] = c_1$ since $f(c_2, y_2) \approx c_1 \in \Delta_f$ and $f(c_2, y_2)$ is the most specific generalization of $f(c_2, c_3)$ among the left-hand sides of equalities in Δ_f. Similarly, we have that $\mathcal{M}[\![f(c_6, c_4)]\!] = c_3$ and $\mathcal{M}[\![f(c_3, c_3)]\!] = c_4$.

Proposition 5

Let M be a T-satisfiable assignment containing only uninterpreted function symbols over uninterpreted sorts. The set Δ constructed by the procedure in Figure 9 is a Σ-map. Moreover, the normal model $\mathcal{M}$ represented by Δ satisfies M.

Proof: To show that Δ is a Σ-map, we show that Σ_f is a defining map for each function symbol f of Σ. Step 1(c) of the procedure ensures that Point 1(b) of Definition 4 is met for all pairs of equalities in Δ_f, while Step 1(d) makes sure that Point 2 is met. We prove by contradiction that Point 1(a) of Definition 4 also holds for Δ_f. Assume that $t \approx v_1, t \approx v_2 \in \Delta_f$ with $v_1 \not\approx v_2$. Due to our construction, both $t \approx v_1$ and $t \approx v_2$ are in $D_1 \cup D_2$. Thus, there must exist terms $t = f(t_1, \ldots, t_n)$ and $s = f(s_1, \ldots, s_n)$ in $\mathbf{T}_M$ such that $\mathcal{V}_M(t_1) = \mathcal{V}_M(s_1)$, …, $\mathcal{V}_M(t_n) = \mathcal{V}_M(s_n)$ and $\mathcal{V}_M(t) = v_1 \not\approx v_2 = \mathcal{V}_M(s)$, contradicting our assumption that M is a (consistent) satisfying assignment. Thus, Δ_f is a defining map for all $f \in \Sigma$, and thus Δ is a Σ-map.

For each term $f(t_1, \ldots, t_n) \in \mathbf{T}_M$, we have that $f(\mathcal{V}_M(t_1), \ldots, \mathcal{V}_M(t_n)) \approx \mathcal{V}_M(t) \in D_1$, and thus $\mathcal{M}(t) = \mathcal{V}_M(t)$. Thus, $\mathcal{M}$ satisfies all equalities between pairs of terms in the same equivalence class of M_c^*. Since M_c^* is T-satisfiable, we have that $\mathcal{M}$ satisfies all disequalities in M_c^* as well. Since M_c^* is a superset of M, we have that $\mathcal{M}$ satisfies M. □

8 Model-based quantifier instantiation

We now focus our attention on Step 3 of our finite model finding procedure $\mathsf{FM\text{-}Solve}_{\mathcal{H}}$ from Figure 3. In this step, the procedure $\mathsf{FM\text{-}Solve}_{\mathcal{H}}$ considers quantified formulas in the set:

$$\{\forall \mathbf{x}\, \varphi \mid (a \Leftrightarrow \forall \mathbf{x}\, \varphi) \in A \text{ and } a \in M\} \tag{8.1}$$

https://doi.org/10.1017/S1471068417000175 Published online by Cambridge University Press

```
proc eval($\mathcal{M}, t, \sigma$) ≡
  match $t$ with
    | $f(t_1, \ldots, t_n)$ → for $j = 1, \ldots, n$
                                 let $(v_j, X_j) = $ eval($\mathcal{M}, t_j, \sigma$)
                               choose a critical argument subset $C$ of $\{1, \ldots, n\}$
                               return $(f^{\mathcal{M}}(v_1, \ldots, v_n), \bigcup_{i \in C} X_i)$
    | $x$ → return $(\sigma(x), \{x\})$
```

Fig. 10. The eval procedure for candidate model $\mathcal{M}$. Returns a pair (v, S) where $(t\sigma)^{\mathcal{M}} = v$, and S is a subset of the domain of σ that was used to compute this interpretation.

Call this set Q. For each formula $\forall \mathbf{x}\, \varphi \in Q$, it uses a quantifier instantiation heuristic $\mathcal{H}$ that returns a set of substitutions from $\mathbf{x}$ to terms in the set $\mathbf{V}$ constructed in Step 2. A trivial way to implement $\mathcal{H}$ is to choose all such possible substitutions. If $\mathbf{x}$ is a tuple of n variables each ranging of a sort with k domain elements, this heuristics will return k^n substitutions, which is clearly unfeasible unless both k and n are rather small. Significantly more scalable heuristics can be adopted if it is possible to identify sets of substitutions σ yielding instances $\varphi\sigma$ that are already satisfied by the current candidate model, as these substitutions can be safely ignored. These heuristics are collectively known as *model-based quantifier instantiation.*

A way to perform model-based quantifier instantiation, as implemented in the SMT solver Z3 (Ge and de Moura 2009), is to use the SMT solver itself as an oracle: a separate copy of the SMT solver is run on another query to determine whether a candidate model $\mathcal{M}$ satisfies each quantified formula. If it does not, a single instance that is falsified by $\mathcal{M}$ is added to the current clause set F. This approach incurs the performance overhead of constructing the corresponding query as well as initializing the oracle. Our version of model-based instantiation relies instead upon specialized data structures when checking candidate models and choosing instantiations, and may add more than one instantiation per invocation.

We describe below a model-based quantifier instantiation method that identifies entire sets of instances as satisfiable in $\mathcal{M}$ without actually generating and checking those instances individually (Reynolds *et al.* 2013b). The main idea is to determine the satisfiability in $\mathcal{M}$ of some instance $\varphi\sigma$ of a quantified formula $\forall \mathbf{x}\, \varphi \in Q$, generalize $\varphi\sigma$ to a set J of instances equisatisfiable with $\varphi\sigma$ in $\mathcal{M}$, and then look for further instances only outside that set. The set J is computed by identifying which variables of φ actually matter in determining the satisfiability of $\varphi\sigma$. Technically, for each $\psi = \forall \mathbf{x}\, \varphi$, substitution $\sigma = \{\mathbf{x} \mapsto \mathbf{v}\}$ into $\mathbf{V}$, and instance $\varphi' = \varphi\sigma$ of ψ, if $\mathcal{M} \models \varphi'$ we compute a partition of $\mathbf{x}$ into $\mathbf{x}_1$ and $\mathbf{x}_2$ and a corresponding partition of $\mathbf{v}$ into $\mathbf{v}_1$ and $\mathbf{v}_2$ such that $\mathcal{M} \models \forall \mathbf{x}_2\, \varphi\{\mathbf{x}_1 \mapsto \mathbf{v}_1\}$; similarly, if $\mathcal{M} \not\models \neg\varphi'$ we compute a partition such that $\mathcal{M} \not\models \forall \mathbf{x}_2\, \neg\varphi\{\mathbf{x}_1 \mapsto \mathbf{v}_1\}$. In either case, we then know that all instances of $\varphi\{\mathbf{x}_1 \mapsto \mathbf{v}_1\}$ over $\mathbf{V}$ are equisatisfiable with φ' in $\mathcal{M}$, and so it is enough to consider just φ' *in lieu* of all them. We will refer to the elements of $\mathbf{x}_1$ above as a set of *critical variables for φ (under σ)*—although strictly speaking this is a misnomer as we do not insist that $\mathbf{x}_1$ be minimal.

https://doi.org/10.1017/S1471068417000175 Published online by Cambridge University Press

8.1 Generalizing evaluations

We have developed a general procedure that, given the Σ-map of a candidate model $\mathcal{M}$, a term t, and a substitution σ over t's variables, computes and returns both the value of $t\sigma$ in $\mathcal{M}$ and a set of critical variables for σ. This procedure effectively extends to quantifier-free formulas as well by treating them as Boolean terms—which evaluate to either true or false in a Σ-interpretation depending on whether they are satisfied by the model or not.

The procedure, called eval, is defined recursively over its input term and is sketched in Figure 10. For uniformity, we assume that function symbols and logical operators are all in prefix form.

When evaluating a non-variable term $f(t_1,\ldots,t_n)$, eval determines a *critical argument subset* C for it. This is a subset of $\{1,\ldots,n\}$ such that the term $f(s_1,\ldots,s_n)$ denotes a constant function in $\mathcal{M}$ where each s_i is the value computed by eval for t_i if $i \in C$, and is a unique variable otherwise. If f is a logical symbol, the choice of C is dictated by the symbol's semantics. For instance, for $\approx(t_1,t_2)$, C is $\{1,2\}$; for $\vee(t_1,\ldots,t_n)$, it is $\{1,\ldots,n\}$ if the disjunction evaluates to false; otherwise, it chooses $\{i\}$ for some i where t_i evaluates to true. If f is a function symbol of Σ, eval computes C by first constructing a custom index data structure for interpreting applications of f to values. The key feature of this data structure is that it uses information on the sets $X_1,\ldots X_n$ to choose an evaluation order for the arguments of f. For example, given the term $t = f(g(x,y,z),v_2,h(x))$, say that eval computes the values v_1,v_2,v_3 and the critical variable sets $\{x,y,z\}$, $\emptyset$, $\{x\}$ for the three arguments of f, respectively. With those sets, it will use the evaluation order $(2,3,1)$ for those arguments—meaning that the second argument is evaluated first, then the third, etc. Using the index data structure, it will first determine if $f(x_1,v_2,x_3)$ has a constant interpretation in $\mathcal{M}$ for all x_1,x_3. If so, then the evaluation of t depends on none of its variables, and the returned set of critical variables for t will be $\emptyset$. Otherwise, if $f(x_1,v_2,v_3)$ has a constant interpretation in M, then the evaluation of t depends on $\{x\}$, or else it depends on the entire variable set $\{x,y,z\}$.

The next example gives more details on the whole process of using eval to generalize a ground instance to a set of ground instances equisatisfiable with it in a given model.

Example 8

Let $Q = \{\forall x_1\, x_2\, f(x_1) \approx g(x_2,b) \vee h(x_2,x_1) \not\approx b\}$, where all terms are of some sort S. Consider a candidate model $\mathcal{M}$ induced by a Σ-map containing the following definitions:

$$\begin{aligned}\Delta_g &= \{g(a,a) \approx c,\ g(y_1,b) \approx a,\ g(y_1,y_2) \approx b\}\\ \Delta_f &= \{f(b) \approx b,\ f(y_1) \approx a\}\\ \Delta_h &= \{h(y_1,y_2) \approx b\}\end{aligned}$$

Suppose $\mathbf{V}_S = \{a,b,c\}$. The table below shows the bottom-up calculation performed by eval on the formula $\varphi = f(x_1) \approx g(x_2,b) \vee h(x_2,x_1) \not\approx b$ with $\mathcal{M}$ above and $\sigma = \{x_1 \mapsto a, x_2 \mapsto a\}$.

https://doi.org/10.1017/S1471068417000175 Published online by Cambridge University Press

input	output	critical arg. subset
x_1	$(a, \{x_1\})$	
x_2	$(a, \{x_2\})$	
b	$(b, \emptyset)$	$\emptyset$
$f(x_1)$	$(a, \{x_1\})$	$\{1\}$
$g(x_2, b)$	$(a, \emptyset)$	$\{2\}$
$h(x_2, x_1)$	$(b, \emptyset)$	$\emptyset$
$f(x_1) \approx g(x_2, b)$	$(\mathsf{true}, \{x_1\})$	$\{1, 2\}$
$h(x_2, x_1) \not\approx b$	$(\mathsf{false}, \emptyset)$	$\{1, 2\}$
$f(x_1) \approx g(x_2, b) \vee h(x_2, x_1) \not\approx b$	$(\mathsf{true}, \{x_1\})$	$\{1\}$

For most entries in the table, the evaluation is straightforward. For a more interesting case, consider the evaluation of $g(x_2, b)$. First, the arguments of g are evaluated, respectively, to $(a, \{x_2\})$ and $(b, \emptyset)$. Using an indexing data structure built from Δ_g for the evaluation order $(2, 1)$, we determine that $g(x_2, b)$ has constant value a for all x_2. Hence, we return an empty set of critical variables for $g(x_2, b)$.

Similarly, the fact that eval returns $(\mathsf{true}, \{x_1\})$ for the original input formula φ and the substitution $\sigma = \{x_1 \mapsto a,\ x_2 \mapsto a\}$ means that we were able to determine that all ground instances of $\varphi\{x_1 \mapsto a\} = (f(a) \approx g(x_2, b) \vee h(x_2, a) \not\approx b)$, not just the instance $\varphi\sigma$, are satisfied in $\mathscr{M}$. We can then use this information in $\mathsf{FM\text{-}Solve}_{\mathscr{H}}$ to completely avoid generating and checking those instances.

8.2 A model-based instantiation heuristic

For any given quantified formula ψ, the eval procedure allows us to identify a set of instances over $\mathbf{V}$ that can be represented by a single one, as far as satisfiability in the candidate model $\mathscr{M}$ is concerned. In this subsection, we present a quantifier instantiation heuristic that generates a set I of instances that together represent *all* instances of ψ over $\mathbf{V}$ that are falsified by $\mathscr{M}$. This kind of exhaustiveness is crucial because it allows us to conclude that $\mathscr{M} \models \psi$ by just checking that I is empty.

The heuristic is implemented by a procedure that relies on eval for computing the set I above, or rather, a set of substitutions for generating the elements of I from ψ. The procedure is fairly unsophisticated and quite conservative in its choice of representative instances, which makes it very simple to implement and prove correct. Its main shortcoming is that it does not take full advantage of the information provided by eval, and so may end up producing more representative instances than needed in many cases.

Let $\psi = \forall \mathbf{x}\, \varphi \in Q$ with $\mathbf{x} = (x_1, \ldots, x_n)$, where Q is the set defined in (8.1). For $i = 1, \ldots, n$, let S_i be the sort of x_i and let $\mathbf{V}_{\mathbf{x}} = \mathbf{V}_{S_1} \times \cdots \times \mathbf{V}_{S_n}$. For each $S \in \{S_1, \ldots, S_n\}$, let $<_S$ be an arbitrary total ordering over the values $\mathbf{V}_S$ of sort S. Let $<$ be the *lexicographic* extension of these orderings to the tuples in $\mathbf{V}_{\mathbf{x}}$ and

https://doi.org/10.1017/S1471068417000175 Published online by Cambridge University Press

```
proc $\mathscr{H}_m(\mathscr{M}, \forall \mathbf{x}\, \varphi)$ ≡
  $I_{\mathbf{x}} := \emptyset$;  $t := \mathbf{v}_{min}$
  do
    $(v, \{x_{i_1}, \dots, x_{i_m}\}) := \mathsf{eval}(\mathscr{M}, \varphi, \{\mathbf{x} \mapsto t\})$
    if $v = \mathsf{false}$ then  $I_{\mathbf{x}} := I_{\mathbf{x}} \cup \{\{\mathbf{x} \mapsto t\}\}$
    $t := \mathsf{next}_i(t)$ where $i = n + 1 - \max\{0, i_1, \dots, i_m\}$
  while $t \neq \mathbf{v}_{min}$
  return $I_{\mathbf{x}}$
```

Fig. 11. A model-based instantiation heuristic $\mathscr{H}_m$, where $\mathbf{x} = (x_1, \dots, x_n)$.

observe that $\mathbf{V}_{\mathbf{x}}$ is totally ordered by $<$. We write $\mathbf{v}_{min}$ to denote the minimum of $\mathbf{V}_{\mathbf{x}}$ with respect to this ordering.

For every $\mathbf{v} = (v_1, \dots, v_n) \in \mathbf{V}_{\mathbf{x}}$, let $\mathsf{next}_i(\mathbf{v})$ denote the smallest tuple $\mathbf{u}$ with respect to $<$ such that $\mathbf{v}(j) <_{S_j} \mathbf{u}(j)$ for some $1 \leqslant j \leqslant n + 1 - i$, if such a tuple exists, and denote $\mathbf{v}_{min}$ otherwise (including when $i > n$)[7].s For instance, with $n = 3$, $S_1 = S_2 = S_3$ and $\mathbf{V}_{S_1} = \{a, b\}$ with $a <_{S_1} b$, we have that $\mathsf{next}_1(a, a, a) = (a, a, b)$, $\mathsf{next}_2(a, a, a) = (a, b, a)$, $\mathsf{next}_2(a, b, a) = (b, a, a)$, $\mathsf{next}_3(a, a, a) = (b, a, a)$, and $\mathsf{next}_2(b, b, a) = \mathbf{v}_{min} = (a, a, a)$. Note that except in the case that $\mathsf{next}_i(\mathbf{v})$ is $\mathbf{v}_{min}$, we have that $\mathbf{v} < \mathsf{next}_i(\mathbf{v})$.

Our instantiation heuristic $\mathscr{H}_m$ is given in Figure 11. It takes in a quantifier-free formula φ with variables $\mathbf{x}$ and returns a set $I_{\mathbf{x}}$ of substitutions σ for $\mathbf{x}$ such that $\mathscr{M} \not\models \varphi\sigma$. At each execution of its loop, the procedure implicitly determines with eval a set I of instances of φ that are equisatisfiable with $\varphi\{\mathbf{x} \mapsto \mathbf{v}\}$ in $\mathscr{M}$, where $\mathbf{v}$ is the tuple stored in the program variable t. The next value t_{next} for t is a greater tuple chosen to maintain the invariant that all the tuples between t and t_{next} generate instances of φ that are in I. To see that, it suffices to observe that these tuples differ from t only in positions that correspond to non-critical variables of φ, namely those before position i where x_i is the first critical variable of φ in the enumeration $x_1, \dots, x_n$. This observation is the main argument in the proof of the following result.

Lemma 3

Let $\mathbf{v}_0, \dots, \mathbf{v}_m$ be all values successively taken by variable t at the beginning of the loop in $\mathscr{H}_m$. Let v_{max} be the maximum element of $\mathbf{V}_{\mathbf{x}}$. Then for all $j = 1, \dots, m$,

(1) $\mathbf{v}_{j-1} < \mathbf{v}_j$,
(2) for all $\mathbf{u}$ with $\mathbf{v}_{j-1} \leqslant \mathbf{u} < \mathbf{v}_j$, $\mathscr{M} \models \varphi\{\mathbf{x} \mapsto \mathbf{u}\}$ iff $\mathscr{M} \models \varphi\{\mathbf{x} \mapsto \mathbf{v}_{j-1}\}$,
(3) for all $\mathbf{u}$ with $\mathbf{v}_m \leqslant \mathbf{u} \leqslant \mathbf{v}_{max}$, $\mathscr{M} \models \varphi\{\mathbf{x} \mapsto \mathbf{u}\}$ iff $\mathscr{M} \models \varphi\{\mathbf{x} \mapsto \mathbf{v}_m\}$.

Proof: (Sketch) The first statement is immediate since for all $j = 1 \dots m$, we have $v_j = \mathsf{next}_k(\mathbf{v}_{j-1})$ for some k and $v_j \neq \mathbf{v}_{min}$. To show the second statement for a j, assume $\mathbf{v}_j = \mathsf{next}_k(\mathbf{v}_{j-1})$ for some k. For each $\mathbf{u}$ where $\mathbf{v}_{j-1} \leqslant \mathbf{u} < \mathbf{v}_j$, we have that $\mathbf{u}(\ell) = \mathbf{v}_{j-1}(\ell)$ for all $\ell \geqslant k$. For all $\ell < k$, the eval procedure determined that the variable x_ℓ was not a critical variable for φ. Since $\mathbf{u}$ and $\mathbf{v}_{i-1}$ vary on only these variables, we have $\mathscr{M} \models \varphi\{\mathbf{x} \mapsto \mathbf{u}\}$ iff $\mathscr{M} \models \varphi\{\mathbf{x} \mapsto \mathbf{v}_{j-1}\}$. The third statement holds for similar reasons as the second. □

[7] Where $\mathbf{v}(j)$ and $\mathbf{u}(j)$ are the jth component of $\mathbf{v}$ and $\mathbf{u}$, respectively.

https://doi.org/10.1017/S1471068417000175 Published online by Cambridge University Press

Theorem 5

The set $I_{\mathbf{x}}$ returned by $\mathscr{H}_m(\mathscr{M}, \varphi, \mathbf{x})$ is empty if and only if $\mathscr{M} \models \forall \mathbf{x}\, \varphi$.

Proof: Thanks to the previous lemma, if there is an instance of φ that is falsified by $\mathscr{M}$, then $\mathscr{H}_m$ will consider at least one $\mathbf{v}_i$ for which $\varphi\{\mathbf{x} \mapsto \mathbf{v}_i\}$ evaluates to false, and hence it will return at least one instance. Conversely, if all instances of φ are satisfied by $\mathscr{M}$, then all instances of φ considered by $\mathscr{H}_m$ evaluate to true, and hence it will return no instances. □

We remark that, for the model finding purposes of procedure $\mathsf{FM\text{-}Solve}_{\mathscr{H}}$, there is no need for the procedure $\mathscr{H}_m$ to compute the full set $I_{\mathbf{x}}$ once it contains at least one substitution. Any non-empty subset would suffice to trigger a (more incremental) revision of the current candidate model $\mathscr{M}$. That said, our current implementation does compute the whole set and adds all the corresponding instances to the clause set F before computing another model for it. Our experiments show that computing and using one substitution at a time is worse for overall performance than computing and using the full set $I_{\mathbf{x}}$.

Example 9

Consider the quantified formula $\forall x_1\, x_2\, \varphi$ and candidate model $\mathscr{M}$ from Example 8. Assume that $a <_S b <_S c$. The result of running $\mathscr{H}_m$ on $\mathscr{M}$, φ and $\mathbf{x} = (x_1, x_2)$ is summarized in the table below. Each row in the column shows the value of variable t at the beginning of the loop in $\mathscr{H}_m$, the result of computing eval, the substitution (if any) added to $I_{\mathbf{x}}$ on that iteration, and the computation of the next tuple of terms $\mathsf{next}_i(t)$.

Iteration	t	$\mathsf{eval}(\mathscr{M}, \varphi, \{\mathbf{x} \mapsto t\})$	Add to $I_{\mathbf{x}}$	i	$\mathsf{next}_i(t)$
1	(a, a)	$(\mathsf{true}, \{x_1\})$	$\emptyset$	2	(b, a)
2	(b, a)	$(\mathsf{false}, \{x_1\})$	$\{x_1 \mapsto b, x_2 \mapsto a\}$	2	(c, a)
3	(c, a)	$(\mathsf{true}, \{x_1\})$	$\emptyset$	2	(a, a)

We begin by setting t to $\mathbf{v}_{min} = (a, a)$. As demonstrated in Example 8, we have that $\mathsf{eval}(\mathscr{M}, \varphi, \{x_1 \mapsto a, x_2 \mapsto a\})$ returns the pair $(\mathsf{true}, \{x_1\})$. The first component of this pair indicates that $\mathscr{M}[\![\varphi\{x_1 \mapsto a, x_2 \mapsto a\}]\!] = \mathsf{true}$, and hence we do not add this substitution to $I_{\mathbf{x}}$. The second component of this pair indicates moreover that this interpretation did not depend on the value of x_2, and hence $(\varphi\{x_1 \mapsto a,\, x_2 \mapsto v\})^{\mathscr{M}} = \mathsf{true}$ for all values of v. Thus, we need not consider $t = (a, b)$ or $t = (a, c)$. Instead, on the second iteration, we consider $\mathsf{next}_2(a, a) = (b, a)$. Subsequently, $\mathsf{eval}(\mathscr{M}, \varphi, \{x_1 \mapsto b, x_2 \mapsto a\})$ returns the pair $(\mathsf{false}, \{x_1\})$. This indicates that $(\varphi\{x_1 \mapsto b, x_2 \mapsto v\})^{\mathscr{M}} = \mathsf{false}$ for all values of v. We add the substitution $\{x_1 \mapsto b, x_2 \mapsto a\}$ to $I_{\mathbf{x}}$ only. Finally, on the third iteration, $\mathsf{eval}(\mathscr{M}, \varphi, \{x_1 \mapsto c, x_2 \mapsto a\})$ returns the pair $(\mathsf{true}, \{x_1\})$; we add no substitutions to $I_{\mathbf{x}}$, and the loop terminates. Overall, $\mathscr{H}_m$ returns the singleton set of substitutions $\{\{x_1 \mapsto b, x_2 \mapsto a\}\}$.

https://doi.org/10.1017/S1471068417000175 Published online by Cambridge University Press

8.3 Enhancement: Heuristic instantiation

Modern SMT solvers rely on syntatic heuristic instantiation methods for finding unsatisfiable instances for quantified formulas (Detlefs *et al.* 2003; de Moura and Bjørner 2007; Reynolds *et al.* 2014). In these methods, quantified formulas are instantiated based on pattern matching. For instance, the solver may choose to instantiate the quantified formula $\forall x\, P(f(x)) \Rightarrow Q(x)$ based on the substitution $\{x \mapsto c\}$ when $P(f(c))$ is a ground term occurring in its current satisfying assignment. This technique is often referred to as *E-matching*. We found that heuristic instantiation-based E-matching can be helpful in the context of our finite model finding approach as well, because the instances it generates are helpful in quickly ruling out candidate models that are obviously spurious.

A quantifier instantiation heuristic $\mathscr{H}$, such as the model-based one from the previous section, can be enhanced by applying heuristic instantiation with a higher priority. That is, we may consider a modified quantifier instantiation heuristic that first computes the set of instances $I_{\mathbf{x}}$ returned by E-matching for a quantified formula $\forall \mathbf{x}\, \varphi$. If this set is non-empty, it returns $I_{\mathbf{x}}$; otherwise, it returns the instances from the original heuristic $\mathscr{H}$ on $\forall \mathbf{x}\, \varphi$.

In practice, we have found that it is best to apply heuristic quantifier instantiation *after* finding a satisfying assignment with a bounded number of equivalence classes. By waiting to apply quantifier instantiation until after a satisfying assignment of this form can be constructed, we can avoid pitfalls common to E-matching. In particular, having a finite cardinality for uninterpreted sorts ensures that only a finite number of terms are unique up to congruence, thus ensuring that E-matching, which is non-terminating in general, will eventually return instances that rule out the current upper bound on cardinality, or terminate with no instances. We discuss the impact of heuristic instantiation further in Section 9.2.

9 Results

We implemented all features mentioned in this paper inside CVC4 (Barrett *et al.* 2011), a state-of-the-art SMT solver based on the DPLL($T_1, \ldots, T_m$) architecture. This section presents experimental results on this implementation[8]. We separate this section into two sets of experiments, the first to evaluate the relative effectiveness of various strategies for the FCC solver, and the second to evaluate the model finder's overall performance when used with quantified formulas. For the second set of experiments, we compare our model finder against state-of-the-art SMT solvers and automated theorem provers.

9.1 FCC *solver evaluation*

We first examine the effectiveness of approach to handling ground problems in the theory of EUF with FCC. In this section, all experiments were run on a Linux

[8] Details can be found at `http://cs.uiowa.edu/~ajreynol/TPLP-fmf`.

https://doi.org/10.1017/S1471068417000175 Published online by Cambridge University Press

machine with an 8-core 2.60 GHz Intel® Xeon® E5-2670 processor with 16 GB of RAM.

We tested various configurations of the FCC solver, starting with the default configuration **cvc4+f**, which contains the region-based enhancements described in Section 6.2, where conflicting states are reported by using clique lemmas of the form $(\neg\mathsf{distinct}(t_1,\ldots,t_{k+1}) \vee \neg\mathsf{card}_{S,k})$. We also tested a configuration, **cvc4+fe**, which reports conflict clauses of the form $(\bar{l}_1 \vee \ldots \vee \bar{l}_n \vee \neg\mathsf{card}_{S,k})$, where $l_1,\ldots,l_n$ are equalities and disequalities that entail $\mathsf{distinct}(t_1,\ldots,t_{k+1})$. This configuration avoids the introduction of new equalities into the search (contained in the expansion of $\mathsf{distinct}$), but has the disadvantage that it can generate different conflict clauses for essentially the same clique. Additionally, we considered configuration **cvc4+f-r**, which differs from **cvc4+f** only in that regionalizations have always just one region per sort S, encompassing the entire disequality graph for S.

We also evaluated the MACE-style approach to finite model finding described in related work (McCune 1994), which we implemented in the configuration **cvc4+mace**. In the case of a set of ground clauses F involving a single sort, if $\mathbf{T}_F$ is the set of all terms in F and $c_1,\ldots,c_k$ are fresh constants serving as domain constants, this configuration checks the satisfiability of

$$F \wedge \mathsf{distinct}(c_1,\ldots,c_k) \wedge \bigwedge_{t\in\mathbf{T}_F} (t \approx c_1 \vee \ldots \vee t \approx c_k) \tag{9.1}$$

for $k = 1, 2, \ldots$ until (9.1) is found satisfiable for some k. Then, the minimal model size for F is k. A major and well-known shortcoming of this approach is the introduction of unwanted symmetries in the problem due to the use of domain constants. CVC4 can address this issue to some extent since it incorporates symmetry breaking techniques directly at the ground EUF level (Déharbe *et al.* 2011).

We considered satisfiable benchmarks encoding randomly generated graph coloring problems and consisting of a conjunction of disequalities between constants of a single sort. In particular, we considered a total of 793 non-trivial problems containing between 20 and 50 unique constants and between 100 and 900 disequalities, and measured the time it takes each configuration to find a model of minimum size, with a 60 second timeout. For the benchmarks we tested, the configuration **cvc4+f** solves the most benchmarks within the time limit: 723. The configuration **cvc4+f** was an order of magnitude faster than **cvc4+fe** on most benchmarks, with the latter only being able to solve 309 benchmarks within the time limit. This strongly suggests that generating explanations for cliques in conflict lemmas involving cardinality constraints is not an effective approach in this scheme.

Figure 12 compares the performance of the configuration **cvc4+f** against **cvc4+f-r** and **cvc4+mace**. The second scatter plot clearly shows that the **cvc4+f** configuration generally requires less time and solves more benchmarks (723 versus 664) than **cvc4+f-r**, confirming the usefulness of a region-based approach for clique detection. The third scatter plot compares **cvc4+f** against **cvc4+mace**. The latter configuration was able to solve only 617 benchmarks and generally performed poorly on benchmarks with larger model size. The median model size of the 123 benchmarks solved only by **cvc4+f** was 17, whereas the median size of the 13 benchmarks solved

https://doi.org/10.1017/S1471068417000175 Published online by Cambridge University Press

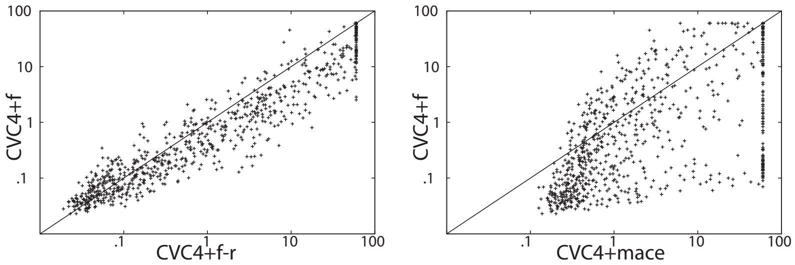

Fig. 12. Results for randomly generated benchmarks. Runtimes are on a log–log scale.

only by **cvc4+mace** was 10. This suggests that for larger cardinalities **cvc4+mace** suffers from the model symmetries created by the introduction of domain constants, something that **cvc4+f** avoids.

9.2 Finite model finder evaluation

We provide results on cvc4 with finite model finding for three sets of benchmarks coming from different formal methods applications, including verification and automated theorem proving. We will refer to various configurations of cvc4 based on the features they include. Configuration **cvc4+f** uses the finite model finding techniques described earlier. Additionally, configurations containing **m** in their suffix use the model-based quantifier instantiation heuristic described in Section 8, and configurations with **i** use heuristic instantiation, which can be paired with finite model finding configurations as described in Section 8.3.

Experiments from Section 9.2.1 were run on a Linux machine with an 8-core 2.60 GHz Intel® Xeon® E5-2670 processor. All others were run on a Linux machine with an 8-core 3.20GHz Intel® Xeon® E5-1650 processor with 16 GB of RAM.

9.2.1 Intel benchmarks

We evaluated the overall effectiveness of cvc4's finite model finder for quantified SMT formulas taken from verification conditions generated by DVF (Goel *et al.* 2012), a tool used at Intel for verifying properties of security protocols and design architectures, among other applications. Both unsatisfiable and satisfiable benchmarks were produced, the latter by manually removing necessary assumptions from verification conditions. All benchmarks contain quantifiers, although only over uninterpreted sorts, and span a wide range of theories, including linear integer arithmetic, arrays, EUF, and algebraic datatypes.

For comparison, we looked at the SMT solvers cvc3 (Barrett and Tinelli 2007) (version 2.4.1)[9], Yices (Dutertre and De Moura 2006) (version 1.0.32), and z3 (De Moura and Bjørner 2008) (version 4.1). We did not consider traditional theorem provers and finite model finders because they do not have built-in support for the

[9] cvc3 is the predecessor of cvc4. The latter was developed from scratch, and does not have any code in common with cvc3.

https://doi.org/10.1017/S1471068417000175 Published online by Cambridge University Press

Sat	**german** (45)		**refcount** (6)		**agree** (42)		**apg** (19)		**bmk** (37)	
	#	time	#	time	#	time	#	time	#	time
cvc3	0	0.0	0	0.0	0	0.0	0	0.0	0	0.0
yices	2	0.0	0	0.0	0	0.0	0	0.0	0	0.0
z3	45	1.1	1	7.0	0	0.0	0	0.0	0	0.0
cvc4+i	2	0.0	0	0.0	0	0.0	0	0.0	0	0.0
cvc4+f	**45**	0.3	**6**	0.1	42	15.5	18	200.0	36	1201.5
cvc4+f-r	**45**	0.3	**6**	0.1	42	18.6	15	364.3	34	720.4
cvc4+fi	45	0.4	**6**	0.1	**42**	14.2	19	492.8	36	831.0
cvc4+fm	**45**	0.3	**6**	0.1	42	23.6	**19**	210.2	37	375.1
cvc4+fmi	**45**	0.3	**6**	0.1	42	16.4	19	221.1	**37**	176.8

Unsat	**german** (145)		**refcount** (40)		**agree** (488)		**apg** (304)		**bmk** (244)	
	#	time	#	time	#	time	#	time	#	time
cvc3	145	0.4	40	0.2	457	6.8	267	77.0	229	76.2
yices	145	1.8	40	7.0	488	1475.4	304	35.8	244	25.3
z3	145	1.9	40	0.9	**488**	10.6	304	12.2	244	5.3
cvc4+i	**145**	0.1	40	0.2	484	6.8	**304**	11.2	**244**	2.9
cvc4+f	145	0.8	40	0.4	476	3782.1	298	2252.5	242	1507.0
cvc4+f-r	145	0.4	40	0.2	475	1574.3	294	3836.0	240	1930.5
cvc4+fi	145	0.7	**40**	0.1	488	188.7	302	342.0	244	660.3
cvc4+fm	145	0.4	40	0.3	471	5018.2	300	1122.7	242	834.1
cvc4+fmi	145	0.3	**40**	0.1	488	185.9	302	339.8	244	668.5

Fig. 13. Number of solved satisfiable and unsatisfiable Intel (DVF) benchmarks and cumulative time for solved benchmarks. All times are in seconds.

theories in our benchmark set. All these solvers use E-matching as a heuristic method for answering unsatisfiable in the presence of universally quantified formulas. Z3 additionally relies on model-based quantifier instantiation techniques to establish satisfiability in the presence of quantified formulas (Ge and de Moura 2009).

The results, separated into unsatisfiable and satisfiable instances, are shown in Figure 13 for five classes of benchmarks and a timeout of 600 seconds per benchmark. The first two classes, **refcount** and **german**, represent verification conditions for systems described in Goel *et al.* (2012); benchmarks in the third are taken from Tuttle and Goel (2012); the last two classes are verification problems internal to Intel.

For the satisfiable benchmarks, our finite model finder is the only tool capable of solving any instance in the last three benchmark classes. In fact, **cvc4+f** is able to solve all but two, and most of them in less than a second. When extended to include techniques for model-based quantifier instantiation (configurations **cvc4+fm** and **cvc4+fmi**), we are able to solve all satisfiable benchmarks within the timeout. By comparing **cvc4+f** against **cvc4+f-r**, we see that the region-based approach for recognizing cliques is beneficial, particularly for the harder classes where the latter configuration solves fewer benchmarks within the timeout. The model sizes found for these benchmarks were relatively small; only a handful had a model with sort

https://doi.org/10.1017/S1471068417000175 Published online by Cambridge University Press

cardinalities larger than 4. To our knowledge, our model finder is the only tool capable of solving these benchmarks.

For the unsatisfiable benchmarks, Yices and Z3 can solve all of them, with Z3 being much faster in some cases. We observe that CVC4 with finite model finding is orders of magnitude slower than the SMT solvers on these benchmarks. This is, however, to be expected since it is geared toward finding models, and applies exhaustive instantiation with increasingly large cardinality bounds, which normally delays the discovery that the problem is unsatisfiable regardless of those bounds.

However, we found that each unsatisfiable problem can be solved by either **cvc4** or **cvc4+fmi**, and in less than 3 seconds. Additionally, configuration **cvc4+fmi** solves all unsatisfiable benchmarks within 900 seconds, suggesting that CVC4's model finder makes consistent progress toward answering unsatisfiable on provable DVF verification conditions. From the perspective of verification tools, the results here seem promising. A common strategy for handling a verification condition would be to first use an SMT solver hoping that it can quickly find it unsatisfiable with E-matching techniques; and then resort to finite model finding if needed to either answer unsatisfiable, or produce a model representing a concrete counterexample for the verification condition.

9.2.2 TPTP benchmarks

We considered benchmarks from a recent version of the TPTP library (Sutcliffe 2009) (5.4.0), a widely used library from the automated theorem proving community. The benchmarks from this library involve no theory reasoning other than equality, and are composed mostly of quantified formulas.

We compared CVC4 (version 1.2) against other SMT solvers including Z3 (version 4.3) and CVC3 (version 2.4.1), as well as various automated theorem provers and model finders for first-order logic, including Paradox (Claessen and Sörensson 2003) and iProver (Korovin 2008) (version 0.99). Paradox is a MACE-style model finder that uses preprocessing optimizations such as sort inference and clause splitting, among others, and then encodes to SAT the original problem together with increasingly looser constraints on the size of the model. iProver is an automated theorem prover based in the Inst-Gen calculus that can also run in finite model finding mode (**iprover+f**). In that mode, it incrementally bounds model sizes in a manner similar to MACE-style model finding. However, it encodes the whole problem into the EPR fragment[10], for which it is a decision procedure. Since these two tools are limited to classical first-order logic with equality, we considered only the unsorted first-order benchmarks of TPTP.

Figure 14 shows results for benchmarks from the TPTP library that are known to be satisfiable or unsatisfiable. All experiments were run with a 300 second timeout per benchmark. The benchmarks were placed into (exactly one) category based on its logical and syntactic characteristics, where EPR includes benchmarks that

[10] This fragment of first-order logic consists of all formulas of the form $\exists \mathbf{x}.\forall \mathbf{y}.\varphi$, where φ is quantifier-free and contains no function symbols.

https://doi.org/10.1017/S1471068417000175 Published online by Cambridge University Press

	Unsat					Sat				
	EPR (920)	**NEQ** (2008)	**SEQ** (7682)	**PEQ** (1796)	**Total** (12406)	**EPR** (388)	**NEQ** (618)	**SEQ** (340)	**PEQ** (612)	**Total** (1958)
z3	840	1406	**3366**	656	6268	345	261	175	160	941
cvc3	596	910	3091	648	5245	24	0	8	0	32
iprover	**888**	**1786**	3346	310	**6330**	**384**	434	106	156	1080
iprover+f	-	-	-	-	-	378	**555**	**224**	268	1425
paradox	-	-	-	-	-	343	534	201	**372**	**1450**
cvc4+i	809	1346	3277	**668**	6100	21	1	8	0	30
cvc4+f	736	900	1261	531	3428	329	441	178	242	1190
cvc4+fm	725	942	1315	419	3401	329	448	214	286	1277
cvc4+fi	733	994	1594	457	3778	329	422	178	231	1160
cvc4+fmi	748	997	1594	459	3798	327	416	190	232	1165

Fig. 14. Number of solved TPTP benchmarks. All experiments were run with a 300 second timeout.

reside in the effectively propositional fragment, NEQ are benchmarks that do not contain any equality reasoning, SEQ are benchmarks containing some equality, and PEQ are benchmarks containing only pure equality. Both configurations **iprover** and **iprover+f** used scheduling strategies that iProver incorporated for CASC 24, a competition for automated theorem provers, meaning that multiple configurations of this solver were run sequentially. The latter of these configurations, as well as the configuration **paradox** were solely run on the satisfiable benchmarks from this set. All configurations of cvc4 with finite model finding used sort inference techniques as described in Reynolds (2013), which is capable of treating unsorted inputs as multi-sorted based on their structure. Sort inference techniques are known to be useful for this set of benchmarks, and are used in most competitive ATP systems, including Paradox and iProver.

For satisfiable benchmarks, cvc4's model finder with exhaustive instantiation (**cvc4+f**) solves 1,190 benchmarks. Using model-based quantifier instantiation, that number goes up to 1,277 (configuration **cvc4+fm**). Using heuristic instantiation (**cvc4+fmi**) in addition to model-based instantiation led to finding fewer satisfiable benchmarks, solving 1,165 within the timeout, suggesting that the solver becomes overloaded with the large number of instantiations produced by exhaustive instantiation.

While cvc4 solves more than z3, which finds 941 satisfiable benchmarks, our model finder still trails the overall performance of the other model finders on these problems. Paradox was the overall best solver, finding 1,450 satisfiable benchmarks. We attribute this to the fact that we have not implemented advanced preprocessing techniques, such as clause splitting, that have been shown to be critical for finding finite models of TPTP benchmarks. Nevertheless, cvc4's model finder solves more satisfiable benchmarks (214) than Paradox for classes of problems having some equality reasoning (SEQ). Collectively, some configuration of cvc4 with finite model finding was able to solve 52 satisfiable benchmarks that **paradox** was not able to solve, and 36 satisfiable benchmarks that **iprover+f** was not able to solve.

Figure 14 also shows results for unsatisfiable problems. Although these results are not comparable to those achieved by state-of-the-art theorem provers, such as Vampire and E (Schulz 2002; Kovács and Voronkov 2013), we note that **iprover**

https://doi.org/10.1017/S1471068417000175 Published online by Cambridge University Press

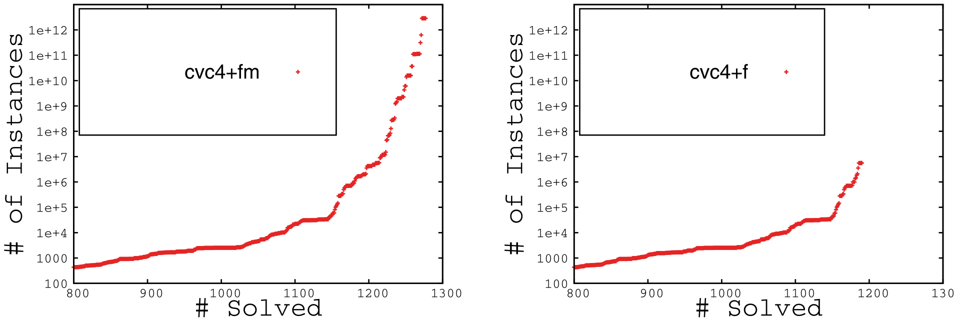

Fig. 15. Satisfiable TPTP problems with (**cvc4+fm**) and without (**cvc4+f**) model-based instantiation. A point (x, y) on this graph says the configuration solves x benchmarks each having at most y ground instances of quantified formulas.

solves the most benchmarks, 6330. Here, **cvc4+fmi** was the best configuration of CVC4 with finite model finding, solving 3,781 within the timeout. While finite model finding configurations solved considerably fewer than using heuristic instantiation alone, some configuration of CVC4 with finite model finding solves 144 unsatisfiable benchmarks that were unable to be solved by any other solver in these experiments, including iProver and Z3.

To further evaluate the impact of model-based quantifier instantiation on our model finder, we recorded statistics on the domain size of quantified formulas in benchmarks solved by its various configurations. We measured the total number of possible ground instances for all quantified formulas in the smallest model for that benchmark (a quantified formula over n variables each with domain size k has k^n instances). For a problem with d total instances, the configuration **cvc4+f** must explicitly generate these d instances, while a model-based configuration may avoid doing so.

The graph on the right-hand side of Figure 15 shows that **cvc4+f** was only able to solve 13 problems having more than 100 K instances, the maximum having around 5.6 million instances. On the other hand, **cvc4+fm** was capable of solving 123 problems having more than 100 K instances, with the largest having more than 2.8 trillion instances. This indicates that the model-based instantiation approach improves the scalability of our model finder, and allows it to solve benchmarks where exhaustive instantiation is clearly infeasible. Model finders such as Paradox have other ways of handling the explosion in the number of instances, namely by minimizing the number of variables per clause. Coupling these techniques with model-based techniques could then lead to additional improvements in scalability. Since techniques for reducing variables in clauses rely on introducing new symbols into the problem, we have found that they have a negative impact on performance for several classes of benchmarks, and thus are disabled by default in CVC4.

9.2.3 Isabelle benchmarks

Recent work has shown that SMT solvers are effective at discharging proof obligations for Isabelle, a generic proof assistant (Paulson and Wenzel 2002).

Sat	Arr	FFT	FTA	Hoare	NSS	QEp	SN	TSq	TSf	Total
z3	2	19	24	47	7	47	1	17	8	172
cvc3	0	9	0	0	0	0	0	8	0	17
cvc4+i	0	9	0	0	0	0	0	8	0	17
cvc4+f	35	145	**177**	162	56	85	12	**57**	90	819
cvc4+fm	33	141	173	155	43	86	12	54	89	786
cvc4+fi	36	146	172	162	**61**	**86**	**12**	55	**93**	823
cvc4+fmi	**36**	**147**	174	**162**	61	83	12	56	93	**824**

Unsat	Arr	FFT	FTA	Hoare	NSS	QEp	SN	TSq	TSf	Total
z3	178	277	917	549	108	325	241	620	**291**	3506
cvc3	**321**	**296**	**1124**	**607**	105	297	207	643	227	3827
cvc4+i	307	288	990	563	**117**	**360**	**242**	**708**	283	**3858**
cvc4+f	165	106	451	239	44	131	88	442	151	1817
cvc4+fm	132	92	442	238	26	160	88	430	128	1736
cvc4+fi	172	185	589	383	47	222	112	585	196	2491
cvc4+fmi	168	186	589	379	47	222	112	584	196	2483

Fig. 16. Number of solved satisfiable and unsatisfiable Isabelle benchmarks for various classes within a 300 second timeout.

The performance of these solvers can benefit from an encoding that makes use of theories (Blanchette *et al.* 2011). We considered a set of 13,041 benchmarks corresponding to both provable and unprovable proof goals, corresponding to a superset of those discussed in Blanchette *et al.* (2011). Most benchmarks in this set contain quantifiers, and a significant portion contain integer arithmetic. For many of them, the quantification is limited to the uninterpreted sorts, thus making our finite model finding approach applicable.

The results are shown in Figure 16. For satisfiable benchmarks, all configurations of CVC4's model finder find more satisfiable problems than Z3, which finds only 172 of them overall. The model-based quantifier instantiation technique from Section 8 (configuration **cvc4+fm**) was less effective than naive instantiation (configuration **cvc4+f**) which solves 819, suggesting that model-based techniques were not effective at minimizing the number of instantiations for this set of benchmarks. Using heuristic E-matching noticeably improved the search for models, as configuration **cvc4+fi** solves 823 satisfiable benchmarks. Using both model-based instantiation and heuristic instantiation, configuration **cvc4+fmi**, found more satisfiable problems (824) than any other configuration.

For unsatisfiable problems, **cvc4+i** is the overall winner, solving 3,858, which was more than both **z3** and **cvc3** which solved 3,506 and 3,827, respectively. Configurations of CVC4 with finite model finding generally solves less unsatisfiable benchmarks, but is orthogonal to other solvers and configurations. In these experiments, 309 unsatisfiable benchmarks that CVC3 cannot solve are solved by at least one configuration of CVC4 with finite model finding. Similarly, a configuration of CVC4 with finite model finding solves 429 unsatisfiable benchmarks that Z3 cannot, and 168 that **cvc4+i** cannot.

https://doi.org/10.1017/S1471068417000175 Published online by Cambridge University Press

10 Conclusion

We developed a general approach for finite model finding in SMT that is efficient for many classes of problems that are of practical interest to formal methods applications. Experimental evidence from an implementation of these methods in the SMT solver CVC4 shows that our approach is effective in practice at solving many classes of benchmarks, including verification conditions from industry, and benchmarks from automated theorem proving libraries. The implementation is highly competitive with respect to other SMT solvers and to automated theorem provers.

In ongoing work, we plan to extend our approach to the problem of finding models of formulas with quantifiers ranging over built-in domains such as the integers and inductive datatypes. We are also investigating the use of CVC4 as a backend to interactive proof assistants such as Isabelle and Coq, where small counterexamples to conjectures are often helpful to the user.

Acknowledgements

We would like to thank Amit Goel and Sava Krstić for their valuable contributions to this work, Sascha Böhme for proving the Isabelle benchmarks, and François Bobot for his help in writing a TPTP front end for CVC4.

References

Baader, F. and Nipkow, T. 1998. *Term Rewriting and All That*. Cambridge University Press.

Barrett, C., Conway, C., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A. and Tinelli, C. 2011. CVC4. In *Proc. of CAV'11*, Lecture Notes in Computer Science, vol. 6806. Springer, 171–177.

Barrett, C., Nieuwenhuis, R., Oliveras, A. and Tinelli, C. 2006. Splitting on demand in SAT modulo theories. In *Proc. of LPAR'06*, Lecture Notes in Computer Science, vol. 4246. Springer, 512–526.

Barrett, C. and Tinelli, C. 2007. CVC3. In *Proc. of the 19th International Conference on Computer Aided Verification (CAV '07)*, W. Damm and H. Hermanns, Eds. Lecture Notes in Computer Science, vol. 4590. Springer-Verlag, Berlin, Germany, 298–302.

Baumgartner, P., Bax, J. and Waldmann, U. 2014. Finite quantification in hierarchic theorem proving. In *Proc. of Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014*, Vienna, Austria, pp. 152–167.

Baumgartner, P., Fuchs, A., de Nivelle, H. and Tinelli, C. 2009. Computing finite models by reduction to function-free clause logic. *Journal of Applied Logic 7* (1), 58–74.

Blanchette, J. C., Böhme, S. and Paulson, L. C. 2011. Extending Sledgehammer with SMT solvers. In *Automated Deduction*, vol. 6803, N. Børner and V. Sofronie-Stokkermans, Eds. Lecture Notes in Computer Science, Springer, 116–130.

Blanchette, J .C. and Nipkow, T. 2010. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In *ITP 2010*, M. Kaufmann and L. C. Paulson, Eds. Lecture Notes in Computer Science, vol. 6172. Springer, 131–146.

Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A. and Sebastiani, R. 2009. Delayed theory combination versus Nelson-Oppen for satisfiability modulo theories: A comparative analysis. *AMAI 55* (1–2), 63–99.

https://doi.org/10.1017/S1471068417000175 Published online by Cambridge University Press

Claessen, K. and Sörensson, N. 2003. New techniques that improve MACE-style finite model building. In *CADE-19 Workshop: Model Computation – Principles, Algorithms, Applications*, 11–27.

de Moura, L. and Bjørner, N. 2007. Efficient E-matching for SMT solvers. In *Proc. of Automated Deduction - CADE-21, 21st International Conference on Automated Deduction*, Lecture Notes in Computer Science, vol. 4603. Springer, Bremen, Germany, 183–198.

De Moura, L. and Bjørner, N. 2008. Z3: An efficient SMT solver. In *Proc. of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08/ETAPS'08*, Springer-Verlag, Berlin, Heidelberg, 337–340.

Déharbe, D., Fontaine, P., Merz, S. and Paleo, B. W. 2011. Exploiting symmetry in SMT problems. In *Proc. of CADE-23*, Lecture Notes in Computer Science, vol. 6803. Springer, 222–236.

Detlefs, D., Nelson, G. and Saxe, J. B. 2003. Simplify: A theorem prover for program checking. *Journal of ACM 52* (3), 365–473.

Dutertre, B. and De Moura, L. 2006. The Yices SMT solver. Version 2.2. Tool paper at http://yices. csl. sri. com/tool-paper. pdf

Garey, M. R., Johnson, D. S. and Stockmeyer, L. 1974. Some simplified np-complete problems. In *Proc. of the 6th Annual ACM Symposium on Theory of Computing, STOC '74*, ACM, New York, NY, USA, 47–63.

Ge, Y., Barrett, C. and Tinelli, C. 2009. Solving quantified verification conditions using satisfiability modulo theories. *Annals of Mathematics and Artificial Intelligence 55* (1–2), 101–122.

Ge, Y. and de Moura, L. 2009. Complete instantiation for quantified formulas in satisfiability modulo theories. In *Proc. of CAV'09,* Lecture Notes in Computer Science, vol. 5643. Springer, 306–320.

Goel, A., Krstić, S., Leslie, R. and Tuttle, M. 2012. SMT-based system verification with DVF. In *Proc. of SMT'12.*

Ihlemann, C., Jacobs, S. and Sofronie-Stokkermans, V. 2008. On local reasoning in verification. In *TACAS 2008*, C. R. Ramakrishnan and J. Rehof, Eds. Springer, Berlin Heidelberg, 265–281.

Jovanovic, D. and Barrett, C. 2013. Being careful about theory combination. *Formal Methods in System Design 42* (1), 67–90.

Korovin, K. 2008. iProver – an instantiation-based theorem prover for first-order logic. In *Proc. of IJCAR'08*, Lecture Notes in Computer Science, vol. 5195. Springer, 292–298.

Kovács, L. and Voronkov, A. 2013. First-order theorem proving and vampire. In *Proc. of Computer Aided Verification - 25th International Conference, CAV 2013*, Saint Petersburg, Russia, 1–35.

Krstić, S. and Goel, A. 2007. Architecting solvers for SAT modulo theories: Nelson-Oppen with DPLL. In *Proc. of FroCoS'07*, Lecture Notes in Computer Science, vol. 4720. Springer, 1–27.

McCune, W. 1994. *A Davis–Putnam Program and its Application to Finite First-Order Model Search: Quasigroup Existence Problems.* Technical Report, Argonne National Laboratory.

Nieuwenhuis, R., Oliveras, A. and Tinelli, C. 2006. 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.

Paulson, L. C. and Wenzel, M. 2002. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, vol. 2283. Springer.

https://doi.org/10.1017/S1471068417000175 Published online by Cambridge University Press

Reger, G., Suda, M. and Voronkov, A. 2016. Finding finite models in multi-sorted first-order logic. In *Proc. of Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference*, Bordeaux, France, 323–341.

Reynolds, A. J. 2013. *Finite Model Finding in Satisfiability Modulo Theories*. PhD Thesis, The University of Iowa.

Reynolds, A., Tinelli, C., Goel, A. and Krstić, S. 2013. Finite model finding in SMT. In *Computer Aided Verification*, vol. 8044, N. Sharygina and H. Veith, Eds. Lecture Notes in Computer Science, Springer, Berlin Heidelberg, 640–655.

Reynolds, A., Tinelli, C., Goel, A., Krstić, S., Deters, M. and Barrett, C. 2013. Quantifier instantiation techniques for finite model finding in SMT. In *Automated Deduction - CADE-24*, M. P. Bonacina Ed. Lecture Notes in Computer Science, vol. 7898. Springer, Berlin Heidelberg, 377–391.

Reynolds, A., Tinelli, C. and de Moura, L. M. 2014. Finding conflicting instances of quantified formulas in SMT. In *FMCAD*, IEEE, 195–202.

Schulz, S. 2002. E–a brainiac theorem prover. *Ai Communications 15* (2, 3), 111–126.

Sutcliffe, G. 2009. The TPTP problem library and associated infrastructure: The FOF and CNF parts, v3.5.0. *Journal of Automated Reasoning 43* (4), 337–362.

Tinelli, C. and Harandi, M. T. 1996. A new correctness proof of the Nelson–Oppen combination procedure. In *Proc. of FroCoS'96*, Applied Logic, Kluwer, Academic Publishers, 103–120.

Torlak, E. and Jackson, D. 2007. Kodkod: A relational model finder. In *Proc. of TACAS'07*, Lecture Notes in Computer Science, vol. 4424. Springer, 632–647.

Tuttle, M. R. and Goel, A. 2012. Protocol proof checking simplified with SMT. In *Proc. of NCA'12*, IEEE Computer Society, 195–202.

Zhang, J. and Zhang, H. 1995. SEM: A system for enumerating models. In *Proc. of IJCAI'95*, 298–303.

https://doi.org/10.1017/S1471068417000175 Published online by Cambridge University Press