Finite Model Finding in SMT[*]

Andrew Reynolds[1], Cesare Tinelli[1], Amit Goel[2], and Sava Krstić[2]

[1] Department of Computer Science, The University of Iowa
[2] Strategic CAD Labs, Intel Corporation

Abstract. SMT solvers have been used successfully as reasoning engines for automated verification. 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 quantified verification conditions. We present a novel finite model finding method that reduces these limitations in the case of quantifiers ranging over free sorts. Our method contrasts with previous approaches for finite model finding in first-order logic by not relying on the introduction of *domain constants* for the free sorts and by being fully integrated into the general architecture used by most SMT solvers. This integration is achieved through the addition of a novel solver for sort cardinality constraints and a module for quantifier instantiation over finite domains. Initial experiments with verification conditions generated from a deductive verification tool developed at Intel Corp. show that our approach compares quite favorably with the state of the art in SMT.

1 Introduction

Techniques and solvers for Satisfiability Modulo Theories (SMT) 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 verification tasks that can be reduced to proving the unsatisfiability of quantifier-free formulas in certain logical theories. A number of verification applications, however, dealing with data structures not modeled by an SMT solver's built-in theories, or analyzing systems with an unbounded number of processes or memory locations, require solvers that can prove the unsatisfiability of *quantified* formulas in those theories.

SMT solvers that can reason about quantified formulas are based on incomplete methods and so often report "unknown" when they fail, after some predetermined amount of effort, to prove a quantified formula unsatisfiable. For verification purposes, however, it is very useful to know when such formulas are indeed satisfiable; especially if the solver can also return some representation of the formula's model, as that can be used to identify errors in the artifact being verified or in the formulation of its intended properties. Current SMT solvers are able to produce models of satisfiable quantified formula only in fairly restricted cases [9], which limits their scope and usefulness.

We reduce these limitations with a novel method for model finding in SMT. By the undecidability of first-order logic there are no automated methods for finding arbitrary

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

models. So we focus on *finite* models, which can be enumerated and represented symbolically. More precisely, since 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 finite model finders for standard first-order logic, the main idea is simply to check universal quantifiers exhaustively over candidate models with increasingly large domains for the free sorts, until an actual model is found. Our method differs from previous approaches by not relying on the explicit introduction of *domain constants* for the free sorts, and by being fully integrated into the general architecture used by many SMT solvers. While our approach is limited to SMT formulas with quantifiers ranging only over free sorts, it is still quite useful because such formulas occur often in verification applications; moreover, when satisfiable they usually have small finite models.

We present our finite model finding method in the context of an abstract framework that models a large class of SMT solvers supporting multiple theories and quantified formulas. An overview of this framework is provided in Section 2. The method itself is described in Section 3. In Section 4, we discuss the initial experimental results obtained with our implementation of the method within the SMT solver CVC4.

Related Work. The state of the art in finite model finding in first-order logic is exemplified by tools such as MACE and Paradox [6]. Their approach is based on encoding to SAT the problem of whether a given set of universally quantified first-order formulas has a model of a given size k. The encoding is based on (i) the introduction of k *domain constants*, fresh constant symbols representing the elements of the model's domain; and (ii) an exhaustive instantiation of the input formulas with these constants. Further ground constraints are added to state that the k domain constants denote all domain elements and are pairwise distinct. The resulting ground formulas are then translated to an equisatisfiable propositional formula and fed to a SAT solver. Advances on this approach mostly focus on addressing two of its main limitations for scalability: the size of the resulting propositional formula and the presence of *value symmetries*, that is, partial solutions that are equivalent modulo a permutation of the domain constants (see [16] for example). One way to drastically reduce the size of the final formula is to encode the problem in a more expressive, but still decidable, fragment first-order logic such as, for instance, function-free clause logic [4].

A completely different approach, pioneered by the SEM model finder [18], is based on traditional constraint satisfaction methods and a built-in treatment of ground equational reasoning. It relies on special search techniques, such as symmetry reduction and least-number heuristics, to enumerate possible models efficiently.

Our method is more similar to SEM-style model finding in that it is not based on reductions to SAT, and is free of the spurious symmetries caused by the use of domain constants. In contrast to SEM, we rely on the DPLL(T) architecture common to many SMT solvers as our core search procedure, and use on-demand instantiation techniques for handling quantifiers. That way, our method can handle natively formulas that also involve operators from theories such as arithmetic, arrays, bit vectors, and so on, as long as none of their quantifiers range over the non-free sorts of these theories.

Formal Preliminaries. We work in the context of many-sorted first-order logic with equality. We fix a set $\mathbf{S}$ of *sort symbols* and for every $S \in \mathbf{S}$ an infinite set of $\mathbf{X}_S$ of *variables of sort* S. We assume the sets $\mathbf{X}_S$ are pairwise disjoint and let $\mathbf{X}$ be their union. A *signature* Σ consists of a set $\Sigma^s \subseteq \mathbf{S}$ of sort symbols and a set Σ^f of *(sorted) function symbols* $f^{S_1 \cdots S_n S}$, where $n \geq 0$ and $S_1, \ldots, S_n, S \in \Sigma^s$. We drop the sort superscript from function symbols when it is clear from context or unimportant. Without loss of generality we use equality, denoted by $\approx$, as the only predicate symbol. We abbreviate $\neg(s \approx t)$ with $s \not\approx t$.

Given a signature Σ, well-sorted terms, atoms, literals, clauses, and (possibly quantified) formulas with variables in $\mathbf{X}$ are defined as usual[1] and referred to respectively as Σ-*terms*, Σ-*atoms* and so on. A *ground term/formula* is a Σ-term/formula with no variables. Where $\mathbf{x} = (x_1, \ldots, x_n)$ is tuple of variables and Q is either $\forall$ or $\exists$, we write $Q\mathbf{x}\,\varphi$ as an abbreviation of $Qx_1 \cdots Qx_n\,\varphi$. If φ is a Σ-formula and $\mathbf{x}$ has no repeated variables, we write $\varphi[\mathbf{x}]$ to denote that φ's free variables are from $\mathbf{x}$; if $\mathbf{t} = (t_1, \ldots, t_n)$ is a term tuple we write $\varphi[\mathbf{t}]$ for the formula obtained from φ by simultaneously replacing each occurrence of x_i in φ by t_i.

A Σ-*interpretation* I maps: each $S \in \Sigma^s$ to a non-empty set S^I, the *domain* of S in I; each $x \in \mathbf{X}$ of sort S to an element $x^I \in I_S$; and each $f^{S_1 \cdots S_n S} \in \Sigma^f$ to a total function $f^I : S_1^I \times \cdots \times S_n^I \to S^I$. A satisfiability relation between Σ-interpretations and Σ-formulas is defined inductively as usual. The *reduct* of I to a sub-signature Ω of Σ is an Ω-interpretation that coincides with I on the symbols in Ω.

A *theory* is a pair $T = (\Sigma, \mathbf{I})$ where Σ is a signature and $\mathbf{I}$ a class of Σ-interpretations, the *models* of T, that is closed under variable reassignment (i.e., every Σ-interpretation that differs from one in $\mathbf{I}$ only for how it interprets the variables is also in $\mathbf{I}$) and isomorphism. A formula $\varphi[\mathbf{x}]$ of T is *satisfiable* (resp., *unsatisfiable*) *in* T if it is satisfied by some (resp., no) interpretation in $\mathbf{I}$. A set Γ of formulas *entails in* T a Σ-formula φ, written $\Gamma \models_T \varphi$, if every interpretation in $\mathbf{I}$ that satisfies all formulas in Γ satisfies φ as well. The set Γ is *satisfiable in* T if $\Gamma \not\models_T \bot$ where $\bot$ is the universally false atom. When Γ and φ are ground we write $\Gamma \models_p \varphi$ if Γ *propositionally entails* φ, that is, if the set $\Gamma \cup \{\neg\varphi\}$ is unsatisfiable when considering all atomic formulas in it as propositional variables. The *combination* $T_1 + T_2$ of two theories $T_1 = (\Sigma_1, \mathbf{I}_1)$ and $T_2 = (\Sigma_2, \mathbf{I}_2)$ is the theory $(\Sigma, \mathbf{I})$ where $\Sigma = \Sigma_1 \cup \Sigma_2$ and $\mathbf{I}$ is the largest class of Σ-interpretations whose reduct to Σ_i is in $\mathbf{I}_i$ for $i = 1, 2$.

2 Satisfiability Modulo Multiple Theories

In its most general formulation, SMT is the problem of determining the satisfiability of a set of formulas in some theory T which is possibly a combination of several theories. Our finite model finding method applies to *lazy* SMT solvers based on the DPLL(T) architecture [12]. Such solvers combine modularly a generic CDCL SAT solver[2] (the *SAT engine*) with one or more reasoners (the *theory solvers*) specialized on deciding the satisfiability of *constraints*, conjunctions of ground literals, in a specific theory. Some SMT solvers are able to reason also about quantified formulas. All of them rely on some form of *heuristic quantifier instantiation* where existential quantifiers are Skolemized and universal ones are instantiated with a heuristic selection of ground terms.

[1] With atoms $s \approx t$ well sorted iff s and t are well sorted terms of the *same* sort.

[2] Conflict-Driven Clause Learning solvers were previously referred to as *DPLL* solvers.

DPLL(T)-style SMT solvers are conveniently described at an abstract level using a rule-based framework introduced by Nieuwenhuis *et al.* [12] and then further developed by Krstić and Goel [11] for solvers that combine multiple built-in theories, and by Ge *et al.* [8] for solvers that use heuristic quantifier instantiation. We synthesize the main ideas of those works in a single framework, focusing on aspects most relevant to our task at hand. We present the general framework here and then show in Section 3 how it can be extended to look for finite models for formulas with quantifiers over free sorts.

Abstract Framework. For the rest of the paper we will consider a theory $T = T_1 + \ldots + T_m$ where each T_i is a theory of signature Σ_i, and T_1 is the theory of equality over "uninterpreted functions", also known as EUF. We call *free* those sort and function symbols whose interpretation is not restricted in any way by any of the theories, and consider them as part of the EUF signature; we call *built-in* all the others. For convenience and without loss of generality, we assume that $\Sigma_1, \ldots, \Sigma_m$ have the same set $\mathbf{S}$ of sort symbols and share a distinguished infinite set $\mathcal{C}_S$ of free constants of sort S for each $S \in \mathbf{S}$. Let $\mathcal{C} = \bigcup_{S \in \mathbf{S}} \mathcal{C}_S$. We also assume that $\mathbf{S}$ includes a Boolean sort Bool and a constant true of that sort—allowing us to encode predicate symbols as function symbols of return sort Bool. As customary, we impose the (real) restriction that the signatures $\Sigma_1, \ldots, \Sigma_m$ share no function symbols besides the constants in $\mathcal{C}$.

We describe SMT solvers for the theory T abstractly as state transition systems. States are either the distinguished state fail or 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 formulas derived from the original input problem, and
- C is either the distinguished value no or 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 and call them the *asserted 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 satisfiability in T: each element of F is either a ground clause or a formula of the form $a \Leftrightarrow \forall \mathbf{x} C[\mathbf{x}]$ where a is a ground atom and C is a clause. Moreover, each atom occurring in F is *pure*, that is, has signature Σ_i for some $i \in \{1, \ldots, m\}$.

Initial states have the form $\langle \emptyset, F_0, \mathsf{no} \rangle$ where F_0 is an input set of formulas to be checked for satisfiability. The expected final states are fail, when F_0 is unsatisfiable in T; or $\langle M, F, \mathsf{no} \rangle$ with M satisfiable in T, F equisatifiable with F_0 in T, and $M \models_{\mathrm{p}} F$.

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.[3] The rules are provided in Figure 1 in *guarded assignment form* [11]. 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 component of the current state. The conclusion describes how each component is changed, if at all. We write $\overline{l}$ to denote the

[3] To simplify the presentation, we do not consider here rules that model the forgetting of learned lemmas and restarts of the SMT solver.

$$\textbf{Decide}\ \frac{l \in \mathrm{Lit}_{\mathsf{F}} \cup \mathrm{Int}_{\mathsf{M}} \quad l,\bar{l} \notin \mathsf{M}}{\mathsf{M} := \mathsf{M} \bullet l} \qquad \textbf{Conflict}_i\ \frac{\mathsf{C} = \mathsf{no} \quad l_1,\dots,l_n \in \mathsf{M} \quad l_1,\dots,l_n \models_i \bot}{\mathsf{C} := \bar{l}_1 \vee \cdots \vee \bar{l}_n}$$

$$\textbf{Fail}\ \frac{\mathsf{C} \neq \mathsf{no} \quad \bullet \notin \mathsf{M}}{\mathsf{fail}} \qquad \textbf{Backjump}\ \frac{\mathsf{C} = l_1 \vee \cdots \vee l_n \vee l \quad \mathsf{lev}\,\bar{l}_1,\dots,\mathsf{lev}\,\bar{l}_n \leq i < \mathsf{lev}\,\bar{l}}{\mathsf{C} := \mathsf{no} \quad \mathsf{M} := \mathsf{M}^{[i]}\, l}$$

$$\textbf{Propagate}_i\ \frac{l_1,\dots,l_n \in \mathsf{M} \quad l_1,\dots,l_n \models_i l \quad l \in \mathrm{Lit}_{\mathsf{F}} \cup \mathrm{Int}_{\mathsf{M}} \quad l,\bar{l} \notin \mathsf{M}}{\mathsf{M} := \mathsf{M}\, l}$$

$$\textbf{Explain}_i\ \frac{\mathsf{C} = l \vee D \quad \bar{l}_1,\dots,\bar{l}_n \models_i \bar{l} \quad \bar{l}_1,\dots,\bar{l}_n \prec_{\mathsf{M}} \bar{l}}{\mathsf{C} := l_1 \vee \cdots \vee l_n \vee D} \qquad \textbf{Learn}\ \frac{\mathsf{C} \neq \mathsf{no}}{\mathsf{F} := \mathsf{F} \cup \{\mathsf{C}\}}$$

$$\textbf{Learn}_i\ \frac{\emptyset \models_i \exists \mathbf{x}\,(l_1[\mathbf{x}] \vee \cdots \vee l_n[\mathbf{x}]) \quad l_1,\dots,l_n \in \mathrm{Lit}_{\mathsf{M}}|_i \cup \mathrm{Int}_{\mathsf{M}} \cup L_i}{\mathsf{F} := \mathsf{F} \cup \{l_1[\mathbf{c}] \vee \cdots \vee l_n[\mathbf{c}]\}}$$

$$\forall\textbf{-Inst}\ \frac{a \in \mathsf{M} \quad a \Leftrightarrow \forall \mathbf{x}\, C[\mathbf{x}] \in \mathsf{F}}{\mathsf{F} := \mathsf{F} \cup \{\neg a \vee C[\mathbf{t}]\}} \qquad \exists\textbf{-Inst}\ \frac{\neg a \in \mathsf{M} \quad a \Leftrightarrow \forall \mathbf{x}\, l_1[\mathbf{x}] \vee \cdots \vee l_n[\mathbf{x}] \in \mathsf{F}}{\mathsf{F} := \mathsf{F} \cup \{a \vee \bar{l}_1[\mathbf{c}],\dots,a \vee \bar{l}_n[\mathbf{c}]\}}$$

Fig. 1. DPLL($T_1,\dots,T_m$) rules. In **Learn**$_i$, $\mathbf{x}$ may be empty. In **Learn**$_i$ and $\exists$**-Inst**, $\mathbf{c}$ are fresh constants from $\mathcal{C}$ of the same sort as $\mathbf{x}$. In $\forall$**-Inst**, $\mathbf{t}$ are ground terms of the same sort as $\mathbf{x}$ and such that $C[\mathbf{t}]$ is in purified form.

complement of literal l and $l \prec_{\mathsf{M}} l'$ to indicate that l occurs before l' in M. The function lev maps each literal of M to the (unique) decision level at which l occurs in M. The set $\mathrm{Lit}_{\mathsf{F}}$ (resp., $\mathrm{Lit}_{\mathsf{M}}$) consists of all ground literals in F (resp., all literals of M) and their complements. For $i = 1,\dots,m$, the set $\mathrm{Lit}_{\mathsf{M}}|_i$ consists of the Σ_i-literals of $\mathrm{Lit}_{\mathsf{M}}$. $\mathrm{Int}_{\mathsf{M}}$ is the set of all *interface literals* of M: the equalities and disequalities between constants c, d with c and d occurring in $\mathrm{Lit}_{\mathsf{M}}|_i$ and $\mathrm{Lit}_{\mathsf{M}}|_j$ for two distinct $i, j \in \{1,\dots,m\}$.

The index i ranges from 0 to m for the rules **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**$_0$, $l_1,\dots,l_n \models_0 l$ simply means that $\bar{l}_1 \vee \cdots \vee \bar{l}_n \vee l \in \mathsf{F}$. Similarly, in **Conflict**$_0$, $l_1,\dots,l_n \models_0 \bot$ means that $\bar{l}_1 \vee \cdots \vee \bar{l}_n \in \mathsf{F}$; in **Explain**$_0$, $\bar{l}_1,\dots,\bar{l}_n \models_0 \bar{l}$ means that $l_1 \vee \cdots \vee l_n \vee \bar{l} \in \mathsf{F}$. The rules **Decide**, **Propagate**$_0$, **Explain**$_0$, **Conflict**$_0$, **Fail**, **Learn**, and **Backjump** model the behavior of the SAT engine, which treats ground atoms as Boolean variables and ignores quantified formulas. The rules **Conflict**$_0$ and **Explain**$_0$ model the conflict discovery and analysis mechanism used by CDCL SAT solvers.

All the other rules but $\forall$**-Inst** and $\exists$**-Inst** 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-constraints in M is unsatisfiable in T_i, or entails some yet undetermined literal from $\mathrm{Lit}_{\mathsf{F}} \cup \mathrm{Int}_{\mathsf{M}}$. In the first case, the sub-solver returns an *explanation* of the unsatisfiability as a conflict clause, which is modeled by **Conflict**$_i$ with $i = 1,\dots,m$. 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 $\mathrm{Int}_{\mathsf{M}}$ in **Decide** and **Propagate**$_i$ achieves the effect of the

Nelson-Oppen combination method [15, 5]. The rule **Learn**$_i$ is needed to model theory solvers following the splitting-on-demand paradigm [3]. When asked about the satisfiability of their constraints, these solvers, may return instead a *splitting lemma*, a formula valid in their theory and encoding a guess that needs to be made about the constraints 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 original formula F_0, which may be generated by such solvers.

The $\forall$-**Inst** and $\exists$-**Inst** rules model the quantifier instantiation mechanism. When the atom a, which serves as a proxy for the quantified formula $\forall \mathbf{x} C$, occurs positively in the current assignment M, the SMT solver adds one or more ground instances of the clause $a \Rightarrow C[\mathbf{x}]$. When a occurs negatively, the system adds the (Skolemized) clause form of $\neg a \Rightarrow \neg\forall \mathbf{x} C$. Instantiation heuristics dictate which instances are generated and how quantifier instantiation applications are interleaved with the other operations.

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

Adapting results from [12, 11, 3], it can be shown that every execution satisfies the following invariants: M contains only pure literals and no repetitions; $\mathsf{F} \models_T \mathsf{C}$ and $\mathsf{M} \models_{\mathrm{p}} \neg\mathsf{C}$ when $\mathsf{C} \neq \mathsf{no}$; every model of T satisfying F satisfies the initial set of formulas. Moreover, in the absence of quantified formulas, 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 fail, the clause set F_0 is unsatisfiable in T. 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 $M \models_{\mathrm{p}} F_0$. With quantified formulas, soundness is preserved but termination and completeness are lost in general.

3 Finite Model Finding with DPLL$(T_1, \ldots, T_m)$

We have developed a method that, given a set F_0 of formulas in purified form, searches for a model of T that satisfies F_0 and interprets all the free sorts as finite sets. Abusing the terminology, we will call such models *finite*.

The basic version of our method is restricted to input sets whose quantified formulas quantify only variables of free sorts. An extended version applies also to quantifiers over built-in sorts such as integer, real, array sorts and so on, as long as the quantified variables occur only as arguments of free function symbols. However, we do not discuss that extension here for space constraints.[4] In the basic version, thanks to the

[4] In fact, the extended version also works with quantifiers over built-in sorts always interpreted as a fixed finite domain such as, for instance, the sorts in the theory of fixed sized bit vectors. However, it is practical only for domains of small size.

use of formulas in purified form, our treatment of terms constructed with built-in function symbols is completely standard: built-in ground literals are processed modularly by their corresponding theory solver, and global consistency of the asserted literals is guaranteed via the exchange of interface literals. As a consequence, we focus on free function symbols here.

We look for finite models with the aid of a new theory FCC (finite cardinality constraints) and a solver for it. We assume FCC is one of the sub-theories $T_1, \ldots, T_m$.

Definition 1 (Theory FCC **of finite cardinality constraints).** *The signature* Σ_{FCC} *of* FCC *consists of (i) the same free sort symbols of* EUF*, (ii) the set* $\mathcal{C}$ *of free constants, and (iii) a constant* $\mathsf{card}_{S,k}$ *of sort* Bool *for each free sort S and integer* $k > 0$*. Its models are all* Σ_{FCC}*-interpretations that satisfy each* $\mathsf{card}_{S,k}$ *exactly when they interpret S as a set of cardinality* $n \leq k$*.*

Note that the only ground atoms in FCC besides those of the form $\mathsf{card}_{S,k}$ are equalities between free constants. It is not difficult to show, using reductions to and from graph coloring, that the satisfiability of ground literals in FCC is an NP-complete problem. A solver for ground EUF constraints and one for ground FCC constraints can be combined Nelson-Oppen style to obtain a solver for *ground* EUF *problems with finite cardinality constraints*. This follows immediately from extended combination results by Ranise *et al.* ([13], Theorems 13 and 21). The main idea is to apply the standard Nelson-Oppen non-deterministic combination procedure [15] but to a *flattened* version of the the original input problem, a set of equational literals with equations and disequation respectively of the form $c \approx f(c_1, \ldots, c_n)$ and $d \not\approx d'$ where f is a symbol of the original problem and $c, c_1, \ldots, c_n, d, d'$ are constants from $\mathcal{C}$. We will call this form a *flat form*. This entails that the theory FCC can be incorporated into our abstract framework without loss of completeness for ground problems, provided that all literals in the problem are in flat form.[5]

3.1 An Efficient Solver for FCC

We have developed an FCC solver meant to be efficient in practice when integrated into the DPLL($T_1, \ldots, T_m$) architecture together with a conventional congruence closure-based solver for EUF. We describe how to use the FCC solver to endow DPLL($T_1, \ldots, T_m$) with finite model finding abilities for EUF in the next subsection. Here, we give a high level overview of the FCC solver and how it cooperates with the EUF solver to solve ground EUF problems with cardinality constraints.

The main idea is first to find a model of the EUF constraints, if it exists; and then try to *shrink* that model as needed to satisfy the cardinality constraints. Consider the constraints $G \cup R$ where

- G is a set of equalities and disequalities in flat form
- R is a set of FCC constraints over the free sorts of EUF
- *any (dis)equality between free constants that occurs in R is also in G*

[5] In reality, a flat form is not needed. One can construct an FCC solver that takes in arbitrary ground EUF literals but treats every EUF term as a constant.

Let $\mathbf{T}_G$ be the set of all (sub-)terms occurring in G. If G is satisfiable, the EUF solver can compute a congruence relation $\equiv_E$ over $\mathbf{T}_G$ that is consistent with G in the sense that $s \equiv_E t$ for all $s \approx t \in E$ and $c \not\equiv_E d$ for all $c \not\approx d \in D$. Since G is in flat form, each equivalence class of $\equiv_E$ of terms of some sort S contains a constant from $\mathcal{C}_S$, so we can use it as the representative of that class and call it a *representative constant for* S. It is well-known that if $c_1, \ldots, c_h$ are all the representative constants for a free sort S, there is an EUF model $\mathcal{M}$ satisfying G such that $S^{\mathcal{M}} = \{c_1, \ldots, c_h\}$ (see, for example, [1] §4.3).

Now consider just the card constraints in R. Since constraints about a sort impose no restrictions on the other sorts in FCC, the FCC solver can look at them separately by sort. So let S be one of the sorts in R and let $K = \{\neg\mathsf{card}_{S,i}\}_{i\in I} \cup \{\mathsf{card}_{S,j}\}_{j\in J}$ collect all the card constraints in R for S. Observe that K is satisfiable in FCC iff $I = \emptyset$ or $J = \emptyset$ or $\max(I) < \min(J)$. When K is satisfiable and J is non-empty, the FCC solver needs to check that $R \cup \mathsf{card}_{S,k}$ is satisfiable with $k = \min(J)$. This is immediate if $k \geq h$ where h is the number of equivalence classes the EUF solver has computed for S. Otherwise, $R \cup \mathsf{card}_{S,k}$ is satisfiable if and only if enough of those classes can be merged to reduce their number to at most k. At this point the FCC solver needs to strengthen R with one equality $c \approx d$ between distinct representative constants. Since $R \cup \{c \approx d\}$ may be unsatisfiable in FCC, or $G \cup \{c \approx d\}$ may be unsatisfiable in EUF, all possible equalities between representative constants may have to be considered. If none of them works, $G \cup \mathsf{card}_{S,k}$ is unsatisfiable. Otherwise, R must be strengthened again as above until S has at most k equivalence classes.

Disequality Graphs. Following the splitting-on-demand approach, the FCC solver will return "satisfiable" or "unsatisfiable" only if it can determine the satisfiability of its constraints without having to guess any equality between representative constants. Otherwise, it will simply identify a possible equality $c \approx d$ and let the SAT engine decide on it by returning the *merge lemma* $c \approx d \vee c \not\approx d$.[6] The solver is able to reduce the number of equality guesses by maintaining a *disequality graph* for each free sort S. This is an undirected graph whose vertices are representative constants for S that occur in G, and whose edges link only vertices c, d with $G \models_{\mathrm{EUF}} c \not\approx d$. This data structure tells the FCC solver that certain pairs of constants, the linked ones, cannot be equated.

We illustrate the overall mechanism and the intended collaboration dynamics between the EUF and the FCC solver with a couple of examples.

Example 1. Let $G \cup R$ be $\{a \approx f(b),\, b \approx f(c),\, a \not\approx b,\, b \not\approx c,\, \mathsf{card}_{S,2}\}$ over the single sort S. From it, the EUF solver computes the congruence $\{\{a, f(b)\}, \{b, f(c)\}, \{c\}\}$. Using a, b, c as the representatives, the FCC solver builds the disequality graph with edges $\{(a,b), (b,c)\}$. Since $\mathsf{card}_{S,2}$ limits the size of S to at most 2, the FCC solver generates the merge lemma $a \approx c \vee a \not\approx c$. Strengthening R with $a \approx c$ produces no EUF conflicts and allows the FCC solver to answer "satisfiable".

Example 2. Consider the constraints $\{c_1 \approx c,\, c_4 \approx c,\, c_1 \not\approx c_2,\, c_2 \not\approx c_3,\, c_3 \not\approx c_4,\, c_4 \not\approx c_5,\, \mathsf{card}_{S,2}\}$ where all the constants have 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 2 elements, and report a

[6] This is slightly inaccurate. In reality, the solver asks the SAT engine to apply **Decide** on $c \approx d$.

conflicting clause consisting of the literals that explain the unsatisfiability: $c_1 \not\approx c \vee c_4 \not\approx c \vee c_1 \approx c_2 \vee c_2 \approx c_3 \vee c_3 \approx c_4 \vee \neg \mathsf{card}_{S,2}$.

3.2 FCC Solver Enhancements: Regions

The FCC solver is able to detect the unsatisfiability of a constraint set $D \cup \{\mathsf{card}_{S,k}\}$, where D is a set of disequalities between representative constants for S, only if the disequality graph corresponding to D contains a $(k+1)$-vertex clique. Now, even just checking for the presence of a $(k+1)$-clique in a n-vertex graph is too expensive in general—as its worst case complexity is $O(n^{k+1}(k+1)^2)$. We have developed a method that reduces that cost in practice by partitioning the vertices of the graph into *regions*.

The partition is updated incrementally as the graph evolves so as to maintain the invariant that any $(k+1)$-clique of the graph is entirely contained in one of the regions. We call such partitions a *regionalization* of the graph. Regionalizations help provide a *weak effort* type of satisfiability check where the theory solver reports "(un)satisfiable" only when the (un)satisfiability of its constraints is immediate, and reports "unknown" otherwise. Frequent weak effort checks are commonly used in SMT solvers [12]. They are useful during the extension of the assignment M, to avoid extensions that are clearly unsatisfiable in one of theories. In contrast, *strong effort checks*, where the theory solver is required to give a definite answer or provide a splitting lemma, are needed (for correctness) only when the SMT solver has found an assignment M that propositionally entails the current set of ground clauses.

Weak Effort Checks. When a vertex is added to the graph it starts into its own singleton region. Two regions are combined into one whenever the addition of an edge or the merging of two nodes breaks the regionalization invariant by creating too many *inter-regional* edges, which link two vertices belonging to different regions. The choice of which regions to combine is made heuristically in an effort to increase the likelihood of generating a $(k+1)$-clique. Specifically, a region is combined with the one with which it shares the highest number of inter-regional edges. For any given regionalization of the disequality graph, *small regions*, those with less than $k+1$ vertices, cannot give rise to a clique. So our solver ignores them and focuses on the *large* regions, the other ones. The solver maintains a set of $k+1$ *watched vertices* from each large region, representing a candidate clique. A new vertex from the region is added to this set whenever two vertices in it are merged, to maintain the set's size at $k+1$.[7] The solver also keeps track of all pairs of watched vertices that are not linked. This way it knows that the region contains a $(k+1)$-clique as soon as the set of those pairs becomes empty. Similarly, it knows that the graph cannot contain any $(k+1)$-cliques as soon as the set of regions reduces to one small region. These facts are used to determine the answer of weak effort satisfiability checks.

Example 3. Consider the constraints $\{c_1 \not\approx c_2, c_2 \not\approx c_3, c_3 \not\approx c_4, \mathsf{card}_{S,2}\}$, all over sort S, and the partition $\{\{c_1, c_2\}, \{c_3, c_4\}\}$. That partition is indeed a regionalization because a 3-clique can span two regions only if it contains two interregional edges, and this

[7] If there are no new vertices to watch, the region has become small and can be ignored.

partition only has one. Adding the disequation $c_2 \not\approx c_4$ or $c_1 \not\approx c_4$, say, breaks the regionalization invariant. In either case, FCC the solver would then merge the two regions (and discover a 3-clique with nodes c_2, c_3, c_4 in the first case).

Strong Effort Checks. When the satisfiability of the current set of FCC constraints cannot be determined immediately as in weak effort checks, the FCC solver looks at ways to reduce the size of large regions by guessing an equality between two unlinked watched vertices in the same region, and returning the corresponding merge lemma. If there are no large regions, it creates one by combining smaller ones heuristically. This process eventually leads to the creation of a clique of watched vertices contradicting the corresponding cardinality constraint, or the creation of a single small region per sort. The solver can then report unsatisfiability in the first case and satisfiability in the second.

3.3 A Finite Model Finding Strategy

In this subsection we show how a DPLL$(T_1,\ldots,T_m)$ solver can be turned into a finite model finder for quantified formulas by incorporating the FCC solver described above and adopting a specific execution strategy.

Suppose T_1 is EUF and T_2 is FCC. The strategy starts by applying the **Propagate**$_i$ rules as much as possible. If propositional or theory conflicts arise in the process, an application of **Conflict**$_i$ and then **Fail** ends the execution. Otherwise, for each free sort S in the input problem, the SMT solver asks the FCC solver for a *cardinality lemma* $\mathsf{card}_{S,k_S} \vee \neg\mathsf{card}_{S,k_S}$, encoding the guessing of a concrete cardinality bound on S and corresponding to one application of **Learn**$_2$. This is followed by a corresponding number of **Decide** applications, asserting the cardinality constraint card_{S,k_s} from each of the new lemmas. From then on, the execution proceeds as usual in DPLL$(T_1,\ldots,T_m)$ solvers, but with no applications of the **∀-Inst** rule. (The **∃-Inst** rule is applied, once, as soon as the proxy literal $\neg a$ gets added to the current assignment.)

In particular, the EUF solver computes the congruence closure of all the EUF equalities in the assignment M and adds to it, with **Propagate**$_1$, all the entailed equalities it derives. The FCC solver builds each disequality graph incrementally, starting with one whose vertices are all the representative constants of the initial EUF equivalence classes. New edges, and possibly new vertices, are added to the graph as disequalities get added to M. Vertices are merged, with each resulting vertex inheriting all the edges of the vertices it replaces, as their equivalence classes get merged by EUF and the corresponding equation between the class representatives is added to M.

As card literals and (dis)equalities between representative constants are added to M and sent to the FCC solver, the solver is asked about the satisfiability of its updated set of constraints. It answers unsatisfiable only if its card constraints are unsatisfiable or they contain a constraint of the form $\mathsf{card}_{S,k}$ and the disequality graph for S contains a $(k+1)$-vertex clique over the representative constants for S. The solver's explanation for the unsatisfiability in the latter case is a lemma of the form $\bigvee_{c\approx d\in E} \neg(c \approx d) \;\vee\; \bigvee_{\neg(c\approx d)\in D} c \approx d \;\vee\; \neg\mathsf{card}_{S,k}$ where E and D are respectively a set of equalities and disequalities in the current assignment M that together generate the clique. Returning that clause corresponds to an application of **Conflict**$_2$, which might lead, through applications of **Explain**$_i$ and then **Backjump**, to alternative identifications of representative constants. If no other alternatives exist, **Backjump** will backtrack to the the decision level right before the addition of $\mathsf{card}_{S,k}$ to M, and assert $\neg\mathsf{card}_{S,k}$. This will cause the FCC solver to consider a larger cardinality k' for S, and return the lemma $\mathsf{card}_{S,k'} \vee \neg\mathsf{card}_{S,k'}$ when queried again.

The constraints given to the FCC solver may be unsatisfiable in the presence of $\mathsf{card}_{S,k}$ even if the disequality graph for S contains no $(k+1)$-cliques—as long as this graph has more than k nodes. To see this, simply observe that satisfiability in this case is akin to the k-colorability problem for that graph, which means that further internal search might be needed to determine satisfiability. As discussed earlier, to keep the FCC solver simple it is enough to have it just determine which pairs c, d of representative constants could be in principle identified in order to shrink the graph, and report that possibility as the merge lemma $c \approx d \vee c \not\approx d$. With a decision strategy that prefers $c \approx d$ over $c \not\approx d$, this lets the SMT solver assert $c \approx d$ with **Decide**. The new literal is then used by the FCC solver to shrink the graph and by the EUF solver to merge the equivalence classes of c and d. Unless this leads to a conflict, the FCC will continue generating merge lemmas until the size of the disequality graph for S goes down to k.

Remark 1. Because of congruence constraints, guesses on merge lemmas may sometimes lead to inconsistencies in EUF, instead of FCC, unless the EUF solver computes and propagates all entailed disequalities—which is usually not the case, for efficiency reasons. For example, suppose the current assignment M is $\{c_3 \approx f(c_1),\, c_4 \approx f(c_2),\, c_3 \not\approx c_4,\, \mathsf{card}_{S,2}\}$ where all the terms have sort S. Unless the EUF solver propagates the entailed literal $c_1 \not\approx c_2$, the FCC solver will construct for S the disequality graph $(\{c_1, \ldots, c_4\}, \{(c_3, c_4)\})$ and may ask the SAT engine to guess $c_1 \approx c_2$. The subset $\{c_3 \approx f(c_1),\, c_4 \approx f(c_2),\, c_3 \not\approx c_4,\, c_1 \approx c_2\}$ of the new assignment will then be found unsatisfiable by the EUF solver. In contrast, guessing for instance $c_1 \approx c_3$ and $c_2 \approx c_4$ will produce a model of the required cardinality. □

When M propositionally entails all the ground clauses in F and all the sub-solvers (including the FCC solver) have reported their constraints to be satisfiable, those clauses have a model that interprets each free sort S as a finite set of representative constants. Following that, the SMT solver goes into an *instantiation round* where it applies the **∀-Inst** rule exhaustively. That is, if for each $a \in \mathsf{M}$ and $a \Leftrightarrow \forall \mathbf{x} C[\mathbf{x}] \in \mathsf{F}$ it adds to F all possible well-sorted instances $C[\mathbf{c}]$, where where the elements of $\mathbf{c}$ are current representative constants. The system processes the new ground clauses, and the new constraints they generate, as described above until **Fail** is applicable or a new model of the (extended set of) ground clauses of F is found. Then the system starts another instantiation round, but adding to F only clause instances $C[\mathbf{c}]$ that had not been added in previous instantiation rounds. More precisely, it will discard a newly generated instance $C[\mathbf{d}]$ if F contains an instance $C[\mathbf{c}]$ where $\mathbf{c}$ and $\mathbf{d}$ are equivalent in the current congruence closure. The whole process stops if a new instantiation round produces no new instances. This is because at that point the literals of M have a finite model that satisfies all the instances of the quantified formulas.

In our default strategy, the cardinality upper bounds expressed with the card constraints start at 1 for each sort and are incremented only by 1 at a time, with the goal of minimizing sort sizes in candidate models. Keeping those sizes small is essential to contain the explosion of formula instances added during instantiation rounds. An additional way to control this explosion is to replace exhaustive instantiation with smarter heuristics that avoid the generation of *redundant* instances of quantified formulas—intuitively, instances guaranteed to be satisfied by the current candidate model. We have developed an effective notion of redundancy and a non-exhaustive instantiation method that relies on advanced data structures to represent and query candidate models. That work and the performance improvements it yields are discussed in a companion paper [14].

Correctness. To argue about the soundness of our finite model finding method (i.e., the input problem is unsatisfiable whenever the execution ends with the fail state), observe first that the method can be described entirely as a particular execution strategy of the abstract framework presented in Section 2. Thus, it suffices to show that the FCC solver is itself sound.

Proposition 1. *Whenever the* FCC *solver returns "unsatisfiable" for a set R of* FCC *constraints, R is unsatisfiable in the* FCC *theory.*[8]

Our model finding method is non-terminating in general because there exists satisfiable quantified EUF formulas with no finite models. The more interesting question is whether it is *finite-model complete*, that is, guaranteed to find a finite model of the input problem if one exists. As described here, our approach is finite-model complete for input problems whose quantifiers all range over the same free sort, but not in the more general case involving several free sorts. Informally, the reason for this incompleteness is that our method allows executions that keep increasing indefinitely the size of the wrong sort. We are working on the definition of a *practical* fairness restriction on executions that addresses this issue. For the sort of applications we have been targeting, however, this source of finite-model incompleteness did not seem to be a problem.

4 Experimental Results

We implemented a finite model finder as described here within CVC4 [2] which is based on DPLL($T_1,\ldots,T_m$). We added to CVC4 our FCC solver, implementing it as a direct extension of CVC4's EUF solver. That solver maintains backtrackable data structures for representing the current congruence closure over EUF terms as well as keeping track of asserted disequalities between them. In addition, the FCC solver maintains data structures for the current regionalization as described in Section 3.2.

We ran 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 compared our model finder against various state-of-the-art SMT solvers, including CVC4 itself. All experiments were run on a Linux machine with an 8-core 2.60GHz Intel® Xeon® E5-2670 processor.[9]

[8] We omit the proof of this proposition because it is relatively straightforward thanks to the restricted cases in which the FCC solver returns "unsatisfiable."

[9] The finite model finder, detailed results, and the non-proprietary benchmarks discussed in this section are available at `http://cvc4.cs.nyu.edu/experiments/CAV-2013/`

4.1 FCC Solver Evaluation

We tested various configurations of the FCC solver, starting with the default configuration **cvc4+f**, which implements the region-based enhancements described in Section 3.2 as well as an additional enhancement where conflict clauses have simply the form $\neg\mathsf{distinct}(c_1,\ldots,c_{k+1}) \vee \neg\mathsf{card}_{S,k}$, where distinct is a variadic logical predicate satisfied exactly when its arguments evaluate to pairwise distinct elements. We also tested a configuration, **cvc4+fe**, where conflict clauses are as described in Section 3.3. This configuration avoids the introduction of new predicates into the search (the distinct ones), 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, which we encoded in the configuration **cvc4+mace**. For a basic idea of this encoding in the simple case of a ground EUF formula φ involving a single sort, if $\mathbf{T}_\varphi$ is the set of all terms in φ and $c_1,\ldots,c_k$ are fresh constants serving as domain constants, this configuration uses CVC4 to check the satisfiability of

$$\varphi \wedge \mathsf{distinct}(c_1,\ldots,c_k) \wedge \bigwedge_{t\in\mathbf{T}_\varphi} (t = c_1 \vee \ldots \vee t = c_k) \tag{1}$$

for $k = 1,2,\ldots$ until (1) is found satisfiable for some k. Then, the minimal model size for φ is k. As we mentioned in Section 1, a major shortcoming of this approach is the introduction of unwanted value symmetries in the problem. CVC4 can address this issue to some extent since it incorporates a few symmetry breaking techniques directly at the EUF level [7].

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, 723, within the time limit. Although not shown here in detail, **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 2 compares the performance of the configuration **cvc4+f** against **cvc4+f-r** and **cvc4+mace**. The first scatter plot clearly shows that the **cvc4+f** configuration generally requires less time and solves more benchmarks (723 vs. 664) than **cvc4+f-r**, confirming the usefulness of a region-based approach for clique detection. The second 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 only by **cvc4+mace** was 10. This suggests

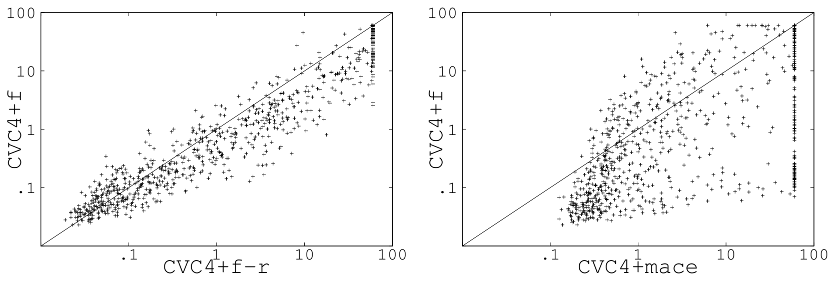

Fig. 2. Results for randomly generated benchmarks. Runtimes are in seconds, on a log-log scale.

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

4.2 Finite Model Finder Evaluation

We also evaluated the overall effectiveness of CVC4's finite model finder for quantified SMT formulas. We used benchmarks derived from verification conditions generated by DVF [10], 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 free sorts, and span a wide range of theories, including linear integer arithmetic, arrays, EUF, and inductive datatypes.

For comparison we looked at the SMT solvers CVC3 (version 2.4.1), Yices (version 1.0.32), Z3 (version 4.1). We also considered CVC4 (release r4751) in native mode, that is, without the finite model finding techniques described here. We did not look at traditional theorem provers and finite model finders because they do not have built-in support for the theories in our benchmark set. All the solvers considered use E-matching as a heuristic method for answering unsatisfiable in the presence of universally quantified formulas. CVC4 uses no sophisticated techniques for detecting satisfiability, which means that it reports "unknown" for most satisfiable quantified problems. In contrast, Z3 additionally relies on model-based quantifier instantiation [9] to be able to detect satisfiable quantified problems in certain cases.

The results, separated into unsatisfiable and satisfiable instances, are shown in Figure 3 for five classes of benchmarks and a timeout of 600s per benchmark. The first two classes, **refcount** and **german**, represent verification conditions for systems described in [10]; benchmarks in the third are taken from [17]; 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.[10] In fact, **cvc4+f** is able to solve all but two, and most of them in less than a second. By comparing **cvc4+f** against

[10] Yices reports "unsat" for two of these benchmarks. We believe that, based on the way they were constructed, the two benchmarks are in fact satisfiable. Also, all other solvers (including previous versions of Yices) time out or answer "unknown".

Sat	**german** (45)		**refcount** (6)		**agree** (42)		**apg** (19)		**bmk** (37)	
	solved	time	solved	time	solved	time	solved	time	solved	time
cvc3	0	0.0	0	0.0	0	0.0	0	0.0	0	0.0
yices	2	0.02	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	2	0.00	0	0.00	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

Unsat	**german** (145)		**refcount** (40)		**agree** (488)		**apg** (304)		**bmk** (244)	
	solved	time	solved	time	solved	time	solved	time	solved	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	**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

Fig. 3. Results for DVF benchmarks. All runtimes are in seconds.

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 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. Interestingly, all of these benchmarks are solved in less than 3s by either **cvc4** (plain CVC4) or **cvc4+f**, indicating that a combination of the two is advantageous in general. We observe that **cvc4+f** is orders of magnitude slower than the SMT solvers on these benchmarks. This is, however, to be expected since it is geared towards 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.

5 Conclusion and Further Work

We presented a method for endowing DPLL(T)-based SMT solvers with finite model finding capabilities for quantified SMT formulas with quantifiers ranging over free sorts. The method relies on a novel and efficient sub-solver for finite cardinality constraints that is fully integrated in the overall SMT solver. Our experimental results with benchmarks generated from a variety of verification applications show that our model finding approach is superior to current quantifier instantiation methods in SMT in the case of satisfiable inputs.

Future work will focus on identifying suitable fair execution strategies that guarantee finite model completeness for problems with multiple free sorts. We are also plan to investigate further approaches for finding models of formulas with quantifiers ranging also over built-in domains such as the integers.

Acknowledgments. We thank Clark Barrett, Morgan Deters, Dejan Jovanović, and François Bobot for their suggestions and assistance in the implementation of our model finder. We thank Mark Tuttle for providing the model from which the **agree** benchmarks were generated.

References

[1] Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998)
[2] Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011)
[3] Barrett, C., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Splitting on demand in SAT modulo theories. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 512–526. Springer, Heidelberg (2006)
[4] Baumgartner, P., Fuchs, A., de Nivelle, H., Tinelli, C.: Computing finite models by reduction to function-free clause logic. Journal of Applied Logic 7, 58–74 (2009)
[5] Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: Delayed theory combination vs. Nelson-Oppen for satisfiability modulo theories: a comparative analysis. Nelson-Oppen for satisfiability modulo theories: a comparative analysis. AMAI 55(1-2), 63–99 (2009)
[6] Claessen, K., Sörensson, N.: New techniques that improve MACE-style finite model building. In: CADE-19 Workshop: Model Computation – Principles, Algorithms, Applications, pp. 11–27 (2003)
[7] Déharbe, D., Fontaine, P., Merz, S., Woltzenlogel Paleo, B.: Exploiting symmetry in SMT problems. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 222–236. Springer, Heidelberg (2011)
[8] Ge, Y., Barrett, C., Tinelli, C.: Solving quantified verification conditions using satisfiability modulo theories. AMAI 55(1-2), 101–122 (2009)
[9] Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 306–320. Springer, Heidelberg (2009)
[10] Goel, A., Krstić, S., Tuttle, R.L.M.: SMT-based system verification with DVF. In: Proceedings of SMT 2012 (2012)
[11] Krstić, S., Goel, A.: Architecting solvers for SAT modulo theories: Nelson-Oppen with DPLL. In: Konev, B., Wolter, F. (eds.) FroCos 2007. LNCS (LNAI), vol. 4720, pp. 1–27. Springer, Heidelberg (2007)
[12] Nieuwenhuis, R., Oliveras, A., Tinelli, C.: 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 (2006)
[13] Ranise, S., Ringeissen, C., Zarba, C.G.: Combining data structures with nonstably infinite theories using many-sorted logic. In: Gramlich, B. (ed.) FroCos 2005. LNCS (LNAI), vol. 3717, pp. 48–64. Springer, Heidelberg (2005)
[14] Reynolds, A., Tinelli, C., Goel, A., Krstić, S., Deters, M., Barrett, C.: Quantifier instantiation techniques for finite model finding in SMT. In: Bonacina, M.P. (ed.) CADE 2013. LNCS, vol. 7898, pp. 377–391. Springer, Heidelberg (2013)
[15] Tinelli, C., Harandi, M.T.: A new correctness proof of the Nelson–Oppen combination procedure. In: Proceeding of FroCoS 1996, pp. 103–120. Kluwer (1996)
[16] Torlak, E., Jackson, D.: Kodkod: a relational model finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007)
[17] Tuttle, M.R., Goel, A.: Protocol proof checking simplified with SMT. In: Proceedings of NCA 2012, pp. 195–202. IEEE Computer Society (2012)
[18] Zhang, J., Zhang, H.: SEM: a system for enumerating models. In: Proceedings of IJCAI 1995, pp. 298–303 (1995)