Finite Model Finding Using the Logic of Equality with Uninterpreted Functions

Amirhossein Vakili and Nancy A. Day

Cheriton School of Computer Science
University of Waterloo
{avakili,nday}@uwaterloo.ca

Abstract. The problem of finite model finding, finding a satisfying model for a set of first-order logic formulas for a finite scope, is an important step in many verification techniques. In MACE-style solvers, the problem is mapped directly to a SAT problem. We investigate an alternative solution of mapping the problem to the logic of equality with uninterpreted functions (EUF), a decidable logic with many well-supported tools (*e.g.,* SMT solvers). EUF reasoners take advantage of the typed functional structures found in the formulas to improve performance. The challenge is that EUF reasoning is not inherently finite scope. We present an algorithm for mapping a finite model finding problem to an equisatisfiable EUF problem. We present results that show our method has better overall performance than existing tools on a range of problems.

1 Introduction

Finite model finding is the problem of finding a satisfying model of a set of first-order logic (FOL) formulas for a finite scope. The utility of finite model finding in verification has been well-established with the popularity of the Alloy Analyzer [11], a tool for writing declarative models in relational algebra, and its Kodkod library for finding satisfying instances [22]. Finite scope analysis has been used in a range of applications, such as code analysis [21], test case generation [12], repairing invalid HTML code [19], temporal logic model checking [23], and counterexample generation for higher-order logic [6].

Approaches to finite model finding have followed two main styles: the MACE-style [14], which reduces the problem to SAT and uses a SAT solver; and the SEM-style [24], which develops an algorithm (usually a backtracking algorithm) for searching for a model explicitly. State-of-the-art tools for model finding are: Kodkod [22], Mace4 [16], and Paradox [8]. Kodkod is a MACE-style solver used in the Alloy Analyzer. Mace4 is used more in the mathematical community and is written in the SEM-style (unlike its predecessor Mace2, which is in the MACE-style). Paradox is a MACE-style solver.

The contribution of our work is the introduction of a new approach to finite model finding in the MACE-style, based on a reduction to the problem of satisfiability in the logic of equality with uninterpreted functions (EUF) [1], and the use of an SMT (satisfiability modulo theories) solver [4]. EUF is many-sorted

(typed), quantifier-free first-order logic with equality. It is a decidable logic and its complexity is NP-complete [1,13]. EUF has advanced solving implementations in many SMT solvers. SMT solvers are first-order logic reasoning tools with an integrated set of decision procedures that use the standard interpretations for various types. We use the SMT solver Z3 [17].

Reynolds *et al.* [18] wrote a SEM-style prover for finite model finding on top of the SMT solver CVC4 [2]. The goal of Reynolds' approach was to find finite satisfying solutions that the SMT solver deemed unsolvable. In our approach, we use the SMT solver directly to solve the whole problem (as in the MACE-style), in contrast to Reynolds approach, which creates a SEM-style solver integrated into the SMT architecture.

As pointed out by Kroening and Strichman [13], despite the fact that the complexity of EUF is the same as propositional logic, there are two reasons to use EUF rather than propositional logic: 1) convenience in modelling, and 2) performance. The larger vocabulary provided by EUF, *i.e.,* equality, uninterpreted functions and types, allows for more concise models. In the approaches that reduce the finite model finding problem to SAT, the structure of types and functions is not well preserved in propositional logic. Since we are reducing the problem to EUF, this structure is retained and exploited in the EUF solving process, which often results in better performance; moreover, translation to EUF eliminates some simplification steps such as term flattening.

The challenge, however, is that problems in EUF are not inherently finite, *i.e.,* the solver does not search only for finite models of a certain scope. To make our approach work, we add *range formulas* that force the solver to consider only instances of a certain finite scope. We re-use many of the techniques found in MACE-style provers, including symmetry breaking, to reduce the model space that must be searched.

The contributions of our work are:

1. Introduction of range formulas to force an EUF solver to search for solutions of an exact scope.
2. A Java library, called Fortress[1], for mapping typed FOL problems (including those specified in the input format TPTP) to SMT-LIB [3] (the standard input language for SMT solvers) for a finite scope.
3. Demonstration that on benchmark problems, overall, Fortress has better performance than Kodkod, Mace4, Paradox, and Reynolds. We show the most improvement on problems that include functions.
4. Demonstration that re-modelling some benchmark problems using the more convenient modelling approach with typed functions results in better performance in Fortress.
5. A comparison of the methods of all the tools to discuss in detail why using an SMT solver is preferable to mapping the problem directly to SAT.

In the next section, we provide some brief background on finite model finding. In Section 3, we show a simple example of how our approach works, and

[1] Available at: `rebrand.ly/fortress`

then we define our translation in Section 4. Section 5 briefly overviews Fortress' implementation. Our results on benchmarks are presented in Section 6. Section 7 demonstrates the advantages of using typed functions, and it is followed by a detailed comparison to related work in Section 8. The conclusion and future work are presented in Section 9.

2 Background

In typed[2] first-order logic (FOL), a *signature* Σ is a pair $\langle \Theta, F \rangle$ where Θ is a set of types, and F is a set of typed functional symbols. Every signature contains the type `Bool`, which represents the Boolean type. A functional symbol $f \in F$ that takes as input n arguments of types $\theta_1, \ldots, \theta_n$ respectively and produces output type is θ is denoted as $f : \theta_1 \times \cdots \times \theta_n \to \theta$. A constant c of type θ, denoted by $c : \theta$, is a functional symbol that has no inputs. In FOL, predicate symbols are functional symbols whose output type is `Bool`. For example, a relational symbol $R : A \times A \to \texttt{Bool}$ denotes a binary relational symbol over type A. Figure 1 shows the rules for constructing the formulas and terms of FOL. The notation $t : \theta$ denotes that the type of the term t is θ. We use this notation only if the type of a term is not obvious from the context.

Formulas

$$
\begin{aligned}
\Phi &::= \top \mid \bot \mid p \\
&::= R(t_1 : \theta_1, \ldots, t_n : \theta_n) \\
&::= \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid \Phi_1 \Rightarrow \Phi_2 \mid \Phi_1 \Leftrightarrow \Phi_2 \\
&::= \exists v : \theta \bullet \Phi \mid \forall v : \theta \bullet \Phi \quad \text{where } v \in V
\end{aligned}
$$

Terms

$$
\begin{aligned}
t : \theta &::= v : \theta \quad \text{where } v \in V \\
&::= c : \theta \\
&::= f(t_1 : \theta_n, \ldots, t_n : \theta_n)
\end{aligned}
$$

Fig. 1. Syntax of FOL over signature $\Sigma = \langle \Theta, F \rangle$ and set of typed variables V, where $c : \theta$, $p : \texttt{Bool}$, $R : \theta_1 \times \cdots \times \theta_n \to \texttt{Bool}$, and $f : \theta_1 \times \cdots \times \theta_n \to \theta$ are in F.

A *structure* (also called a *model* or an *instance*) M over a signature $\Sigma = \langle \Theta, F \rangle$ is a pair $\langle \mathcal{U}, .^M \rangle$, where $\mathcal{U}$, the universe of M, is a collection of mutually disjoint non-empty sets, and $.^M$ is a mapping with the following properties:

1. for each θ in Θ, $\theta^M \in \mathcal{U}$,
2. for each two distinct θ_1 and θ_2, $\theta_1^M \cap \theta_2^M = \emptyset$,
3. for each $p : \texttt{Bool}$ in F, $p^M \in \{\texttt{True}, \texttt{False}\}$,
4. for each $R : \theta_1 \times \cdots \times \theta_n \to \texttt{Bool}$, R^M is a subset of $\theta_1^M \times \cdots \times \theta_n^M$,
5. for each $c : \theta$ in F, $c^M \in \theta^M$,
6. for each $f : \theta_1 \times \cdots \times \theta_n \to \theta$, f^M is a total function from $\theta_1^M \times \cdots \times \theta_n^M$ to θ^M.

We assume the standard semantics for FOL, and use $M \models \Phi$ to denote that M is a structure that *satisfies* the formula Φ [10], meaning that Φ is true

[2] We use "type" and "sort" interchangeably in this paper.

in structure M. We also use the notation $M \models \Gamma$, where Γ is a set of FOL formulas, to denote that M satisfies all the formulas in Γ.

Given a set of FOL formulas Γ over signature $\Sigma = \langle \Theta, F \rangle$, and a function `bounds` from Θ to natural numbers, the *finite model finding problem* means determining if Γ has a satisfying structure M in which for every θ in Θ, the size of the set assigned to θ by M is the finite number $\texttt{bounds}(\theta)$ (*i.e.,* $|\theta^M| = \texttt{bounds}(\theta)$). For each type θ, $\texttt{bounds}(\theta)$ is called the *size of the scope* or just the scope. M is finite because the types in M are each of a fixed, finite, known size. In untyped FOL, there is only one type and therefore only one scope is relevant.

The *logic of equality with uninterpreted functions*, EUF, is a subset of FOL without quantifiers and variables, that includes the equality predicate (usually written in infix form) with its standard interpretation. Checking whether a finite set of EUF formulas has a satisfying structure is decidable and its complexity is NP-complete [13].

3 Small Example

In this section, we present a small example to illustrate the challenge in mapping the finite model finding problem to EUF.

Suppose $\Sigma = \langle \{A, B\}, \{f : A \to B\} \rangle$ is a signature, and we are given the following formula:

$$\forall x, y : A \bullet \ f(x) = f(y) \ \Rightarrow \ x = y \tag{1}$$

The functional symbol f maps elements of A to B and the formula in Equation 1 states that every element of A is mapped to a unique element of B; in other words, no two distinct elements of A are mapped to the same element of B. We are interested in checking if this formula has a model where the size of A is 3 and the size of B is 2. Equation 1 means that in every finite model, the size of B must be greater or equal to the size of A; therefore, there is no model with the scopes proposed. To reduce this problem to checking the satisfiability of a set of EUF formulas, we introduce three new constant symbols of type A, a_1, a_2, a_3, two new constant symbols of type B, b_1, b_2, and generate a set of constraints stating that these new constants are distinct:

$$\{a_1 \neq a_2, \ a_1 \neq a_3, \ a_2 \neq a_3, \ b_1 \neq b_2\} \tag{2}$$

Using the introduced constants, we expand each quantifier by substituting the new constants for the variables. This step generates a set of EUF formulas:

$$\{f(a_i) = f(a_j) \Rightarrow a_i = a_j \mid 1 \leq i, j \leq 3\} \tag{3}$$

If we pass the formulas in Equations 2 and 3 to an EUF solver, such as an SMT solver, and check for their satisfiability, the solver finds a satisfying model where B has three elements, rather than B having two elements as required. This example shows that expanding quantifiers is not sufficient to reduce the finite model finding problem to EUF satisfiability checking. One might think a

remedy to this problem is by adding a formula that states the only members of B are b_1 and b_2:

$$\forall b : B \bullet \ b = b_1 \vee b = b_2$$

This formula has a quantifier, therefore it is not part of EUF and its universal quantifier needs to be expanded with b_1 and b_2:

$$b_1 = b_1 \vee b_1 = b_2, \quad b_2 = b_1 \vee b_2 = b_2$$

This formula is a tautology and therefore, adding it has no effect.

Our solution to this problem is as follows: instead of adding a constraint that ensures B has only two elements, we add constraints, which we call *range formulas*, that guarantee the "effect" of B having two elements. In this example, the effect of B having two elements is that for all $a : A$, $f(a)$ must be either b_1 or b_2:

$$\forall a : A \bullet \ f(a) = b_1 \vee f(a) = b_2 \tag{4}$$

Expanding this equation results in the following set of EUF formulas:

$$\{f(a_i) = b_1 \vee f(a_i) = b_2 \mid 1 \leq i \leq 3\}$$

An EUF solver shows that this set of formulas along with the constraints of Equations 2 and 3 are unsatisfiable.

A range formula for a functional symbol ensures that an EUF solver does not generate an instance that is outside the provided scope:

Definition 1. *For a finite type $\theta = \{e_1, .., e_n\}$ and a functional symbol $f : T_1 \times .. \times T_m \to \theta$, the following is the range formula that we add to ensure that the values assigned to f by an EUF solver are within the specified scope of θ:*

$$\forall v_1 : T_1, .., v_m : T_m \bullet \ f(v_1, .., v_m) = e_1 \vee .. \vee f(v_1, .., v_m) = e_n$$

4 Translation to EUF Logic

Suppose Γ is a set of FOL formulas over signature $\Sigma = \langle \Theta, F \rangle$, and `bounds` is a function from Θ to natural numbers. The finite model finding problem means determining if Γ has a finite model M where for each type θ in Θ, the size of θ^M is equal to $\texttt{bounds}(\theta)$. Our translation to EUF consists of four steps:

1. Normalize each formula in Γ
2. Generate the universe
3. Add range formulas
4. Ground each normalized formula

Step (3) is the main novel contribution of our paper along with the idea of using EUF solvers in the MACE-style for the finite model finding problem. For the other steps leading up to EUF, we borrow the best practices from existing solvers and include their description here for completeness. Next, we explain each step in detail and illustrate the translation using the following example:

Example 1. Let $\Sigma = \langle \{A\}, \{f : A \to A\} \rangle$ be a signature. We want to check if the following two formulas have a model where the size of A ($\texttt{bounds}(A)$) is 3 by translating it to an equisatisfiable set of EUF formulas.

1. $\forall x, y : A \bullet \ f(x) = f(y) \ \Rightarrow \ x = y$
2. $\exists y : A \bullet \ \forall x : A \bullet \ f(x) \neq y$

The first constraint states that f is a one-to-one mapping from A to itself. The second constraint states that the range of f is a proper subset of A. These two formulas are only satisfiable by an infinite model since it is not possible to have a one-to-one mapping from a finite set to one of its proper subsets.

Step 1 - Normalize. The normalization step consists of the following transformations: 1) put each formula in prenex normal form, and 2) skolemize and remove existential quantifiers. Applying these transformations to the formulas of Example 1 results in the following two formulas:

1. $\forall x, y : A \bullet \ \neg(f(x) = f(y)) \vee x = y$
2. $\forall x : A \bullet \ f(x) \neq sk$

In the second formula, sk is a constant of type A that is introduced as the result of skolemization. After normalization, each formula is either quantifier-free or it is of the following form $\forall x_1 : \theta_1, \ldots, x_n : \theta_n \bullet \ \Psi$, where Ψ is quantifier-free. The complexity of this step is linear with respect to the size of the FOL formulas.

Step 2 - Generate Universe. In this step, for each type θ in Θ, we generate $\texttt{bounds}(\theta)$ constants of type θ, and we assert that these constants are mutually distinct. The generated constants at this step are fresh, do not appear anywhere in Γ, and constitute the universe. In Step 3 (adding range formulas), the fact that the introduced constants do not appear in Γ allows us to generate optimized range formulas based on symmetry breaking.

In Example 1, we declare constants a_1, a_2, a_3 of type A and add a constraint to ensure that these constants are mutually distinct. The complexity of this step is linear with respect to the size of the provided bounds[3].

Step 3 - Add Range Formulas: EUF solvers check for the satisfiability of a set of quantifier-free formulas without putting any restrictions on the number of elements assigned to each type. To ensure that the elements of a type θ in a model generated by an EUF solver are exactly the ones declared in Step 2, we add range formulas for constants and functional symbols stating that their values must be equal to the elements of the universe of that type. As mentioned in Section 3, the range formulas allow us to reduce the finite model finding problem to EUF solving. The complexity of adding range formulas is exponential with respect to the arity of the functional symbols.

In Example 1, the following are the range constraints:

$$sk = a_1 \vee sk = a_2 \vee sk = a_3, \quad f(a_1) = a_1 \vee f(a_1) = a_2 \vee f(a_1) = a_3,$$
$$f(a_2) = a_1 \vee f(a_2) = a_2 \vee f(a_2) = a_3, \quad f(a_3) = a_1 \vee f(a_3) = a_2 \vee f(a_3) = a_3$$

[3] In SMT-LIB, this constraint is written simply as: `(distinct` a_1 a_2 a_3`)`.

We use Claessen and Sörensson's symmetry breaking technique [8] to reduce the number of range formulas needed. Since the values a_1, a_2, and a_3 do not appear in the original formulas, one can assume an ordering on them and reduce the range formulas to the following:

$$\begin{aligned} sk &= a_1, \\ f(a_1) = a_1 \vee f(a_1) &= a_2, \\ f(a_2) = a_1 \vee f(a_2) = a_2 \vee f(a_2) &= a_3, \\ f(a_3) = a_1 \vee f(a_3) = a_2 \vee f(a_3) &= a_3 \end{aligned}$$

where the first term is required to be a certain constant and the subsequent terms have gradually more freedom in their possible values. Using symmetry breaking to reduce the number of range formulas does not reduce the complexity of this step.

Step 4 - Ground Formulas. The last step of our translation is grounding: instantiating each universally quantified formula with the generated universe of Step 2. As we substitute different constants for variables that are universally quantified, we immediately simplify the generated formulas based on literals that are discovered and the fact that the elements of the universe are mutually distinct. For example, in the formula $\forall x, y : A \bullet\ f(x) \neq f(y) \vee x = y$, when x and y are substituted with a_1, the generated formula $f(a_1) \neq f(a_1) \vee a_1 = a_1$ is simplified to $\top$ and it is discarded. Also, when x is substituted with a_3 and y with a_2, the generated formula $f(a_3) \neq f(a_2) \vee a_3 = a_2$ is simplified to $f(a_3) \neq f(a_2)$ since we know that $a_2 \neq a_3$. Moreover, we have a syntactic ordering on formulas where $t = s$ is considered to be the same as $s = t$ for any two terms. This ordering allows us to remove some redundant formulas that are generated during the grounding step. The result of grounding Example 1 is the following set of formulas:

$$\begin{gathered} f(a_1) \neq f(a_2), \quad f(a_1) \neq f(a_3), \quad f(a_2) \neq f(a_3), \\ f(a_1) \neq sk, \quad f(a_2) \neq sk, \quad f(a_3) \neq sk \end{gathered}$$

The complexity of this step is exponential with respect to the number of nested universal quantifiers.

We omit the proof that checking the satisfiability of the generated EUF formulas from Steps 3 and 4 is equivalent to checking if the original FOL formulas have a finite model where the size of each type θ is $\texttt{bounds}(\theta)$ since it is quite straightforward.

5 Implementation

Fortress is a Java library for creating typed first-order logic formulas and producing finite model finding problems in SMT-LIB based on the translation of Section 4. Besides the API, we parse a subset of TPTP. In Fortress, formulas are represented as typed lambda calculus terms and all type checking is done at

this level for generality. Once type checked, FOL terms are converted to a more compact representation suitable for FOL.

There are two types of simplifications/optimizations that can be applied: 1) simplifications on FOL terms not specific to finite model finding, such as positive and negative propagations [16], 2) optimizations specific to finite model finding, such as symmetry breaking constraints [8]. Since SMT solvers do an excellent job at type 1 above, these are not implemented in Fortress. However, since SMT solvers do not treat uninterpreted types as finite sets, optimizations of type 2 are implemented in Fortress. We have flags to enable symmetry breaking and our experiments have shown that SMT solvers cannot infer symmetries for finite scope analysis and therefore, they need to be explicitly implemented.

6 Results

We compared our approach to Kodkod (version 2.1 with Minisat), Alloy (version 4.2 with Minisat), Mace4, and Paradox (version 4). We used Z3 (version 4.4.2) as our backend EUF solver for Fortress. We compared the performance of the tools on a set of TPTP benchmarks [20] that were originally used by Torlak and Jackson in [22]. We tested on increased scopes for some benchmarks compared the results reported in [22]. Fortress accepts TPTP as input. Torlak and Jackson had manually translated TPTP examples to Kodkod and we used their translated versions when comparing to Kodkod. Paradox accepts TPTP as input, and Mace4 comes with a tool (tptp_to_ladr [15]) that translates TPTP to its input format. To compare with Alloy, we developed a simple translator from TPTP to Alloy. We included Alloy in this comparison because it is equivalent to using Kodkod without special support for partial instances (see Section 8).

All of these benchmarks are unsat: they do not have finite models with respect to the provided scope sizes. Unsatisfiable cases are better for the comparison of different tools because they are usually much harder than satisfiable ones. These benchmark problems are all untyped and some contain functions.

Table 1 presents the performance of all tools. For Fortress, the performance numbers include both the time for translation and the time for solving by Z3. All our experiments were run on an Intel®Core^{TM}i7-3667U machine running Ubuntu 14.04 64-bit with up to 7.5GB of user memory. We used the solvers in their default mode, without any flags or a customized configuration. Entries marked by "–" indicate the analyses that did not finish within 1800 seconds (30 minutes). The shaded entries show the fastest solver for each benchmark (based on all scopes considered); where the difference was negligible we shaded the entries for multiple tools.

The last three rows of Table 1 summarize the performance of the solvers: Fortress produced the best results more often than any other tool. We also added up the performance time for all the benchmark problems. In this summation, we counted timeouts as 1800 seconds, which is preferential to all the other solvers since Fortress produced results without timing out on all benchmark problems. The total time for Fortress was 2504 seconds. The total times for Kodkod, Alloy,

Table 1. Benchmark Problems (Time in Seconds)

	Scope Size	Fortress	Kodkod	Alloy	Mace4	Paradox
alg195	14	1	0	30	–	5
alg197	21	1	0	20	–	5
num378	21	2	0	–	0	6
infinity	5	0	1	1	0	0
	15	0	19	57	0	0
	25	0	704	–	0	0
alg212	6	0	5	3	0	0
	8	8	207	201	1	5
	10	563	–	–	6	81
com008	7	4	0	0	–	0
	9	48	0	0	–	0
	11	335	1	4	–	58
geo091	7	3	2	12	–	7
	9	9	29	33	–	279
	11	24	745	268	–	–
geo158	7	3	1	1	–	80
	9	9	28	17	–	–
	11	24	378	233	–	–
med009	7	2	0	0	19	0
	9	11	0	0	141	0
	11	31	0	0	139	0
num374	5	2	21	20	0	3
	6	38	262	358	6	147
	7	850	–	–	613	–
set943	7	1	4	66	–	55
	9	2	–	–	–	–
	11	2	–	–	–	–
set948	7	1	0	71	–	62
	9	2	0	–	–	–
	11	4	1	–	–	–
top020	7	2	1	2	0	0
	8	13	4	8	0	1
	9	509	13	16	0	17
Best out of 13		7	6	1	5	2
Total Time		2504	9637	15821	31525	15211
Total Time X		1X	3.85X	6.32X	12.59X	6.07X
		Fortress	Kodkod	Alloy	Mace4	Paradox

Mace4, and Paradox are respectively 3.85, 6.32, 12.59, and 6.07 times the total time of Fortress. This shows that, overall, Fortress is significantly better than the state-of-art solvers.

We also ran the benchmarks using the tool of Reynolds *et al.* [18], however since their tool solved only 2 of the 33 benchmark problems within the 30 minutes time threshold, its results are not presented in Table 1.

A closer look at the benchmarks show that Fortress excels at solving problems that have functional symbols, such as `geo091` and `set943`. Also, SMT solvers are capable of using terms with functions to simplify the reasoning steps by rewriting equalities, such as those found in `alg195` and `num378`. In some cases, this rewriting can solve the problem without performing any search.

Next, we compared the performance of multiple SMT solvers as backends for Fortress. We compared the performance of Z3, CVC4 (version 1.4), and MathSAT5 (version 5.3.10) [7] on six of the nontrivial benchmarks. Table 2 presents the time that it took for each SMT solver to check the satisfiability of the SMT-LIB models generated by Fortress. Our results show that Z3 is more effective in solving EUF formulas that are generated as the result of finite model finding than CVC4 and MathSAT5.

Table 2. Comparing SMT solvers (Time in Seconds)

	`alg212`	`com008`	`geo091`	`med009`	`num374`	`top020`
Scope Size	10	9	11	11	6	8
Z3	562	47	6	6	38	11
CVC4	–	69	45	53	–	3
MathSAT5	–	91	7	20	117	3

7 Exploiting Functions and Types

Functions vs. Relations. Functions and relations have the same expressive power: a total function $f : A \to B$ can be described as a relation $R_f : A \times B \to$ `Bool` with the following two constraints:

$$\forall a : A \bullet \exists b : B \bullet \ R_f(a, b), \tag{5}$$

$$\forall a : A,\ b, b' : B \bullet \ b = b' \vee \neg R_f(a, b) \vee \neg R_f(a, b') \tag{6}$$

where Constraint 5, a *totality definition*, states that every element of A is mapped to some element of B and Constraint 6, a *functional definition*, states that every element of A is not mapped to more than one element of B. Every relation is also a function: a relation maps every tuple to `True` or `False`, depending on if the tuple is in the relation or not. Kodkod and Paradox consider functions as relations accompanied by the totality and functional definitions. Since functions

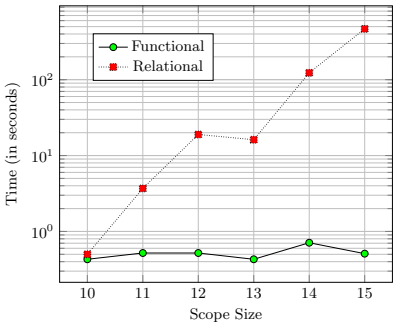

Fig. 2. Lists: Functions vs. Relations

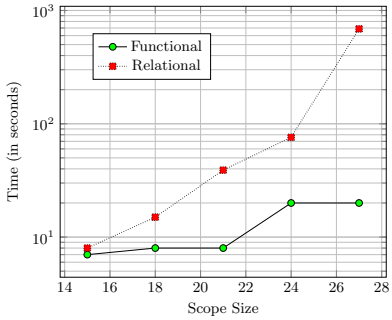

Fig. 3. `MED009`: Partitioning Attributes

are built into EUF, Fortress does not need to add the totality and functional definitions, which simplifies the translation.

Another important benefit of functions is that they allow "true" skolemization. Skolemization is a technique to remove existential quantifiers by introducing functions. For example, in the formula $\forall a : A, b : B \bullet \exists c : C \bullet P(a, b, c)$, skolemization results in the introduction of a functional symbol $sk : A \times B \to C$ and the formula $\forall a : A, b : B \bullet P(a, b, sk(a, b))$. In a language where functions are considered as relations, the skolem function sk needs to be accompanied by the totality definition $\forall a : A, b : B \bullet \exists c : C \bullet R_{sk}(a, b, c)$, which still has an existential quantifier.

To see the effect of using functions on the performance of the SMT solvers for finite model finding, in Fortress we modelled a simple theory of lists presented in [11] in both the functional and relational styles. Figure 2 compares the performance of Fortress for both approaches on different scopes. As depicted in this plot, functions improve the performance of Fortress. For the relational approach, the performance degrades rapidly as the scope size increases. For example, for the scope size 15, the relational approach takes over 7 minutes whereas in the functional approach the scope size of 30 is analyzed in less than 10 seconds (not shown on the plot).

Types. In an untyped system, all elements are in one set. For example, to model a database system for a university, `Person`, `Courses`, `IDs`, *etc.,* are entities that need to be modelled. In an untyped relational world, all these are in one set, and any mapping from one set to another, such as `id` : `Person` $\to$ `IDs`, becomes a relation that is only defined for people and needs totality and functional definitions. In typed systems, types *partition* the universe into subsets. These partitions have two benefits for finite model finding: 1) functions from one type to another can be defined succinctly, 2) in the grounding step (Step 4), a universal quantifier is only expanded for elements of the relevant type.

Together, functions and types can lead to concise modelling of some concepts. For example, in an untyped, relational language, to state that each `Person` in a university is either a `student`, `faculty`, or a `staff` member, three unary relations over the type `Person` must be declared. Four FOL constraints are re-

quired to express that these unary relations partition the set `Person`: every person belongs to one of the partitions, and three other constraints that ensure that no one belongs to more than one category. In a language with types and functions, the same concept can be modelled by introducing a new type `Role` with three elements `student`, `faculty`, `staff`, and introducing a function `attribute` : `Person` $\rightarrow$ `Role`. The totality and functional properties of `attribute` ensure that at least one role is assigned to each person and no one is assigned more than one role respectively. We call the values of the type `Role` *partitioning attributes*. In the relational style, the number of FOL formulas that are required to model partitioning attributes with N values is $\binom{N}{2} + 1$, which is quadratic with respect to the number of values. A functional approach eliminates the need for these constraints.

To evaluate the effect of using types and functions for partitioning, we manually translated a modified version of `med009` and compared the untyped, relational version to one with partitioning via types and functions, and compared the results. Figure 3 shows that performance of the functional approach is much better than the relational approach in Fortress.

8 Comparison with Related Work

In this section, we discuss the question of why our method of using EUF to solve FOL problems of finite scope has better overall performance than related solvers. First, we briefly present the method of each related solver and then present a number of points of comparison. Table 3 summarizes the options and methods supported by different finite model finders.

Table 3. Comparison of Finite Model Finders

	Fortress	Kodkod	Paradox	Mace4	Reynolds
Solver	SMT	SAT	SAT	SEM	SEM/SMT
Input	TPTP, Java API	Java API	TPTP	LADR, TPTP	SMT-LIB
Types	YES	NO	NO	NO	YES
Functions	YES	NO	YES	YES	YES
Relational Ops	NO	YES	NO	NO	NO
Symmetry Breaking	Static	Static	Static	Dynamic	EUF
Partial Instances	NO	YES	NO	NO	NO

8.1 Related Solvers

Kodkod [22] is the MACE-style solver used in the Alloy Analyzer. Its Java API accepts untyped FOL formulas with relational constructs, such as join and transitive closure, as input. Functions must be transformed into relations having functional properties prior to using Kodkod. Once bounds are provided, Kodkod transforms transitive closure into a finite number of applications of join. Kodkod translates the finite model finding problem to SAT using the following steps: 1) detect symmetries in the model and compute symmetry breaking predicates, 2)

allocate Boolean variables to represent relations, 3) expand quantified formulas and make them into constraints over the allocated Boolean variables, and 4) transform the generated Boolean constraints to CNF form. Kodkod represents relations by sparse matrices of Boolean variables, and some of the relational operations become matrix operations. To simplify the translated formulas, Kodkod represents expanded quantified formulas as Compact Boolean Circuits (CBCs). This representation allows Kodkod to detect sharing structures in the grounded formulas and as a result, produce a more optimized CNF formula. Kodkod optimizes for explicitly provided partial instances by using this information during the translation to CNF step.

Paradox [8] is MACE-style prover, whose first step is to allocate a set of Boolean variables to represent each functional symbol. These Boolean variables encode each functional symbol as a relation. Then, every formula is "flattened": a process that removes nested function applications in a formula. For example, flattening the formula $\forall x \bullet f(g(x)) = x$ results in the formula $\forall x, t \bullet\ g(x) = t \Rightarrow f(t) = x$. At this point, the quantifiers of the given formulas are instantiated with all possible values from the universe resulting in a set of quantifier-free formulas. Each of these quantifier-free formulas are translated to propositional logic using the allocated Boolean variables. Since functional symbols are encoded as relations, Paradox adds "functional definition" constraints (every input is mapped to at most one value), and "totality definition" constraints (every input is mapped to some value). The result of this translation is a CNF formula that is passed to a SAT solver. To improve its performance, Paradox uses three techniques: 1) reduce the number of nested quantifiers by splitting disjunctions, 2) adding symmetry breaking constraints, and 3) inferring sorts (types) from the formulas to optimize the translation to SAT.

Mace4 [16] is a SEM-style finite model finder: it has its own backtracking search mechanism to try different assignments. To check if a set of FOL formulas has model of size n, Mace4 allocates "cells" that range from 0 to $n-1$ for each functional symbol. By skolemizing, every existential quantifier is removed. After skolemization, the universal quantifiers are expanded using the elements of $\{0, .., n-1\}$. The expanded formulas are now constraints over the allocated cells. The search mechanism assigns values to cells and checks if the assignment contradicts any of the expanded formulas. If a contradiction is detected, it backtracks; otherwise, the search goes on until either all cells are assigned or there is no possible assignment left. Mace4 uses the least number heuristic to detect some symmetries [25]. It also has a propagation mechanism that allows the search algorithm to prune its search tree.

Reynolds et al. [18] extended CVC4 with finite model finding capabilities so that for satisfiable instances of undecidable SMT logics a user could get a finite model and the SMT solver would not report "unknown". They combined finite model finding with decision procedures for built-in theories using the DPLL(T) architecture. Their approach does not introduce constants and can be classified as a SEM-style technique. According to our results, the method of Reynolds is

not effective in finding finite models of a specific size when SMT theories are not used and the problem is unsatisfiable.

8.2 Comparison

Types. Since EUF is typed, in Fortress we benefit from types without requiring any special mechanism to infer sorts as is done in Paradox. In an untyped language, types can be mimicked by predicates and this is the approach used to translate problems in the typed Alloy language to Kodkod. Fortress' direct use of sorts can reduce the number of constraints generated in the quantifier expansion step because only elements of the correct sort are substituted into the formula for the quantified variable.

Functions. Kodkod does not support functions and assumes that functions have been converted to relations. In Fortress, we do not need to flatten the functional symbols or add the special functional definition and totality constraints required by Paradox since we are translating to a logic that includes functional symbols. As a result, there are fewer constraints in our representation in EUF and in that of Reynolds. Furthermore, the SMT solver can exploit the structure of these functions in its reasoning; in particular, terms that contain functional symbols are used for rewriting and simplification of the input problem. Examples of such techniques are *(near) assignment* and *(near) elimination* simplifications [16].

Relational Operators. The relational operators of Kodkod (e.g., join) do not increase its expressive power but they ease the modelling task. In FOL, the meaning of such operators can be represented through logical operations and quantifiers.

Symmetry Breaking Predicates. In most MACE-style finite model finders, such as Fortress, Kodkod, and Paradox, symmetry breaking is *static*: a set of constraints are added to the model to prevent the solver from exploring symmetric instances. Such constraints are called symmetry breaking predicates. Fortress uses the same symmetry breaking predicates as Paradox. In SEM-style finite model finding, the symmetry detection is built into the search algorithm and it is performed dynamically during the model finding stage [24].

Partial Instances. A partial instance for a set of FOL formulas is an explicit assignments of values to some variables. Kodkod supports explicitly provided partial instances. The first three case studies, `alg195`, `alg197`, and `num378` contain partial instances. Fortress and Kodkod outperform other tools on these case studies. Alloy uses Kodkod as its solver, and yet its performance on the first three case studies is not comparable to Kodkod because the partial instances are not explicitly given to Kodkod. In Fortress, we get good performance without any explicit support for partial instances. Partial instances in EUF are regular constraints that happen to be equalities of variables to values. SMT solvers have sophisticated mechanisms to propagate equalities and reduce the constraint solving time.

Exact scopes. Currently, we only support analysis for a fixed scope, whereas in the Alloy Analyzer, a scope can be specified to include all instances of a certain

size or smaller. Kodkod has this capability and it encodes the whole problem in one SAT formula. On the other hand, Mace4 has an iterative approach that solves each fixed scope separately.

Transitive Closure. Because the Alloy language includes the second-order transitive closure operator, Kodkod supports it and expands its definition using a brute force method (for a finite scope). Our method and the other solvers do not currently support transitive closure, but it would be straightforward to add a step to expand the transitive closure operator as is done in Kodkod.

8.3 Other Related Work

Baumgartner et al. reduce the finite model finding problem to function-free clause logic [5]. Similar to Kodkod, they represent functions as relations with functional constraints. As their results show, current function-free clause logic reasoners are not efficient enough. According to the authors, their results are "as good as" Paradox. We were not able to access their tool.

Elghazi and Taghdiri [9] translate Alloy to SMT-LIB to provide analysis of unbounded scopes. Alloy is translated to an undecidable logic, and SMT solvers are considered as FOL theorem provers that do not necessarily terminate.

9 Conclusion and Future Work

In this paper, we have shown that by reducing the finite model finding problem to the logic of equality with uninterpreted functions (EUF), we can use an SMT solver to find instances with better performance than existing approaches based on translations of the problem to SAT. In our translation, we add range formulas to force the SMT solver to search only for models of a finite scope. Our results show that maintaining the structure of problems (in this case, the types and function structure) can be beneficial in analysis procedures that need to explore exhaustively a model space (as opposed to flattening the problem before search). Our results also give credit to the excellent development of tools of the SMT-solver community.

With respect to modelling constructs, we would like to integrate with Alloy and extend our method to handle the transitive closure operator and a range of scopes. We are also considering taking SMT-LIB as input and the specification of scopes and creating SMT-LIB output. The challenge here is that we do not support all of SMT-LIB, *i.e.,* all of its built-in types (such as taking a finite scope for reals).

In the future, we plan to automate the inference of functional patterns, such as the partitioning attributes in Section 7, to improve the performance of Fortress. Also, our benchmarks show that despite the fact that our technique for finite model finding is superior to the state-of-the-art, there are some benchmarks that other tools solve faster than Fortress. We plan to explore a characterization of the problems that different methods are good at and create a *portfolio* solver for finite model finding.

References

1. Ackermann, W.: Solvable Cases of the Decision Problem. North Holland Publishing Company (1954)
2. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanovi, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Computer-Aided Verification (CAV), LNCS, vol. 6806, pp. 171–177. Springer (2011)
3. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.5. Tech. rep., Department of Computer Science, The University of Iowa (2015)
4. Barrett, C., Sebastiani, R., Seshia, S., Tinelli, C.: Satisfiability Modulo Theories, Frontiers in Artificial Intelligence and Applications, vol. 185, chap. 26, pp. 825–885. IOS Press (2009)
5. Baumgartner, P., Fuchs, A., de Nivelle, H., Tinelli, C.: Computing finite models by reduction to function-free clause logic. Journal of Applied Logic 7(1), 58 – 74 (2009)
6. Blanchette, J.C., Nipkow, T.: Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In: Interactive Theorem Proving (ITP). pp. 131–146. Springer (2010)
7. Cimatti, A., Griggio, A., Schaafsma, B., Sebastiani, R.: The MathSAT5 SMT Solver. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 7795. Springer (2013)
8. Claessen, K., Sörensson, N.: New techniques that improve mace-style finite model finding. In: Proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications (2003)
9. El Ghazi, A.A., Taghdiri, M.: Analyzing Alloy constraints using an SMT solver: A case study. In: International Workshop on Automated Formal Methods (2010)
10. Fitting, M.: First-Order Logic and Automated Theorem Proving. Springer (1990)
11. Jackson, D.: Software Abstractions - Logic, Language, and Analysis. MIT Press (2012)
12. Khurshid, S., Marinov, D.: TestEra: Specification-based testing of Java programs using SAT. Automated Software Engineering 11(4), 403–434 (2004)
13. Kroening, D., Strichman, O.: Decision Procedures: An Algorithmic Point of View. Springer (2008)
14. McCune, W.: A Davis-Putnam program and its application to finite first-order model search: Quasigroup Existence Problem. Tech. rep., Argonne National Laboratory (1994)
15. McCune, W.: Prover9 and mace4 (2005–2010), `http://www.cs.unm.edu/~mccune/prover9`
16. McCune, W.: Mace4 reference manual and guide. CoRR cs.SC/0310055 (2003)
17. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 4963, pp. 337–340. Springer (2008)
18. Reynolds, A., Tinelli, C., Goel, A., Krstić, S.: Finite Model Finding in SMT. In: Computer-Aided Verification (CAV). LNCS, vol. 8044, pp. 640–655. Springer (2013)
19. Samimi, H., Schfer, M., Artzi, S., Millstein, T., Tip, F., Hendren, L.: Automated repair of HTML generation errors in PHP applications using string constraint solving. In: International Conference on Software Engineering (ICSE). pp. 277–287 (2012)

20. Sutcliffe, G.: The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. Journal of Automated Reasoning 43(4), 337–362 (2009)
21. Taghdiri, M., Jackson, D.: Inferring specifications to detect errors in code. Automated Software Engineering 14(1), 87–121 (2007)
22. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 4424, pp. 632–647 (2007)
23. Vakili, A., Day, N.A.: Temporal logic model checking in Alloy. In: International Conference on Abstract State Machines, Alloy, B, VDM, and Z (ABZ). LNCS, vol. 7316, pp. 150–163. Springer (2012)
24. Zhang, H., Zhang, J.: MACE4 and SEM: A comparison of finite model generators. In: Automated Reasoning and Mathematics - Essays in Memory of William W. McCune. pp. 101–130 (2013)
25. Zhang, J., Zhang, H.: Sem: A system for enumerating models. In: International Joint Conference on Artificial Intelligence (IJCAI)