Experiments on Infinite Model Finding in SMT Solving

Julian Parsert[1], Chad E. Brown[2], Mikoláš Janota[2], and Cezary Kaliszyk[3]

[1] University of Edinburgh, Edinburgh and University of Oxford, Oxford, UK
julian.parsert@gmail.com
[2] Czech Technical University in Prague, Prague, Czech Republic
mikolas.janota@cvut.cz
[3] University of Innsbruck, Innsbruck, Austria and INDRC, Czech Republic
cezary.kaliszyk@uibk.ac.at

Abstract

We propose infinite model finding as a new task for SMT-Solving. Model finding has a long-standing tradition in SMT and automated reasoning in general. Yet, most of the current tools are limited to finite models despite the fact that many theories only admit infinite models. This paper shows a variety of such problems and evaluates synthesis approaches on them. Interestingly, state-of-the-art SMT solvers fail even on very small and simple problems. We target such problems by SyGuS tools as well as heuristic approaches.

1 Introduction

Satisfiable problems find a wide range of applications in formal methods and automated reasoning. Notably, a satisfiable problem represents a counterexample to an invalid conjecture. In automated reasoning such instances are often referred to as *counter-satisfiable* (TPTP [44] includes `CounterSatisfiable` as a problem status). But satisfiability problems are also interesting for researchers that look for specific objects, e.g. in computational algebra. This paper is concerned with the resolution of satisfiable problems in the context of *satisfiability modulo theories* (SMT) [12].

SMT solvers are extremely powerful on satisfiable ground problems but much weaker when it comes to problems including quantifiers and uninterpreted functions. One possible approach is *finite model finding*, which has a long-standing history in theorem proving. MACE4 [32] dates to 2003 and is still popular, especially among researchers in computational algebra [23]. Other techniques for finite model finding use SAT solvers, constraint programming and SMT solvers, cf. Sec. 7. By design, these approaches fail on problems that only admit infinite models.

Model-guided quantifier instantiation [24] (implemented in Z3 [22]) enables the construction and checking of infinite models in the presence of general quantifiers. However, this technique alone, does not give a recipe for creating models as it is primarily focuses refutation. Further techniques to construct models in saturation-based theorem provers were proposed [34], but we are not aware of any mainstream prover providing such models (this also requires the prover to saturate, which alone occurs rarely in practice). Some model finding techniques rely on special forms of models, such as linear algebra [26].

Contributions: In this paper, we use synthesis approaches to construct infinite models. At the formal level, the two tasks are identical but practically, the objectives imply very different problems and solutions. In particular:

- We present a number of small problems for which infinite models exist, but there are no finite ones or the finite ones are too large (Sec. 3). These include manually constructed problems and problems from proof assistant hammer systems.

- We experiment with current SyGuS techniques on SMT problems (Sec. 4) and propose simple heuristics that let us extend the range of solvable problems (Sec. 5).
- Our evaluation (Sec. 6) shows that synthesis only enables solving a limited range of problems, while our heuristic solves some problems with unknown status.

2 Syntax-Guided Synthesis

In general, program synthesis is undecidable. It therefore makes sense to consider fragments of this problem. Such fragments can be defined by restrictions on the semantics as well as the syntax of possible solutions. One such approach is *Syntax-Guided Synthesis* [3,5] (SyGuS). The SyGuS input format (SyGuS-IF)[1] [3] is a standardised language that allows users to formulate synthesis problems restricting the problem syntactically (by providing a grammar) and semantically (by providing a formal specification). Much like in SMT, SyGuS problems are stated in the context of a theory (e.g. LIA, EUF, etc.). Naturally, the syntax of SyGuS-IF very closely follows that of SMTLIB [11]. Formally, a SyGuS problem is defined as follows:

Definition 1. *A SyGuS problem is a tuple $\langle T, F, \phi, G \rangle$ where T is a background theory, F is a second-order variable, ϕ is a specification using F as well as symbols from T, and G is a grammar producing a language in the theory T. A solution e to a SyGuS problem is an expression such that $T \models \phi[F/e]$ and $e \in L(G)$ where $L(G)$ is the language generated by G.*

In the following example we present a synthesis problem with a corresponding SyGuS-IF syntax.

Example 1. *Consider the following SyGuS problem $\langle LIA, \text{max2}, \phi, G \rangle$ where ϕ is*

$$\text{max2}(x, y) \geq x \wedge \text{max2}(x, y) \geq y \wedge (\text{max2}(x, y) = x \vee \text{max2}(x, y) = y)$$

and G is

$$I \to x \mid y \mid 0 \mid 1 \mid I + I \mid I - I \mid (\text{ite}[2] BII)$$
$$B \to B \wedge B \mid B \vee B \mid \neg B \mid I = I \mid I <= I \mid I >= I.$$

This describes a SyGuS problem in linear integer arithmetic (LIA) where specification ϕ describes the maximum of the two argumnets to max2. *The grammar consists of two non-terminals I and B mapping to expressions of type integer and boolean respectively. Using the SyGuS-IF format, the same problem can be stated as follows:*

```
(set-logic LIA)
(synth-fun max2 ((x Int) (y Int)) Int
  ((I Int) (B Bool))
  ((I Int (x y 0 1 (+ I I) (- I I) (ite B I I)))
  (B Bool ((and B B) (or B B) (not B) (= I I) (<= I I) (>= I I)))))

(declare-var x Int)
(declare-var y Int)

(constraint (>= (max2 x y) x))
```

[1] https://sygus.org/

[2] ite is the if-then-else operator.

```
(constraint (>= (max2 x y) y))
(constraint (or (= x (max2 x y)) (= y (max2 x y))))
(check-synth)
```

The following expression would be a candiate solution:

```
(define-fun max2 ((x Int) (y Int)) Int (ite (<= x y) y x))
```

A SyGuS problem is essentially an existentially quantified second-order sentence. A theoretical analysis of the decidability and complexity of SyGuS fragments is presented in [19]. Modern SyGuS tools such as EU Solver [4], CVC4/cvc5 [39], and ConcSolver [29] incorporate a vast range of techniques such as: Counterexample guided inductive synthesis (CEGIS) [1], refutation based synthesis, single invocation synthesis [40], unification based synthesis [6] etc.

It is important to point out that a lot of important research in program synthesis also happens outside of context of SyGuS. For example, *Escher* [2] is an algorithm for synthesising recursive programs by interacting with the user via input-output examples. Programming-by-example (PBE) has seen a lot of development with practical applications in spreadsheets with Flashfill and subsequent work [18, 36]. Synthesis has also been applied to programming languages such as Scala [31]. Furthermore, Sketch [43] allows for the automated completion of "holes" in procedural programming languages.

3 Applications

This section presents a number of problems that only admit infinite models. We only discuss the problems that we have been able to construct and experimented with. A more systematic analysis of problems that only admit infinite or extremely large models is left as future work.

Infinite Sorts There are many ways how natural numbers can be axiomatized, however, none of them admit finite characterisations. This implies that many satisfiable problems that include natural numbers require infinite models. As the integers naturally include the natural numbers, this is of course also applicable to theories that include integers, such as LIA [11]. Likewise, any sort and collection of assertions from which one can construct the natural numbers as a predicate over the sort can only be satisfied by taking the sort to be infinite.

A set (or sort) A is said to be *Dedekind-infinite* if there is an injective function from A into A that is not surjective. This is, for example, used as the axiom of infinity on the base type of individuals in the standard library of the HOL Light theorem prover [27]. The type of natural numbers can then be constructed as the smallest type that contains the zero (one of the elements not in the image of the injection) and is closed under successor (given by the injection) on such individuals. Our example `infin0` is satisfiable by taking the declared sort S to be an infinite set and declared function f to be injective but not surjective.

Another way to state that a set (or sort) is infinite is to assert the existence of an irreflexive transitive serial relation. For example, Andrews uses this as an axiom of infinity in his formulation of higher-order logic [7]. Our example `infin1` is satisfiable by any (necessarily infinite) sort and appropriate relation on the sort. Our example `infin2` replaces the sort by Int and the relation by $>$ (clearly irreflexive, transitive and serial).

Algebras and Datatypes Recursive datatypes are a feature of SMT that also has applications, mostly in functional programming [33] and theorem proving [13]. Showing the satisfiability of properties that concern lists, streams, infinite trees and alike often requires infinite models. We include only basic problems that require the synthesis of infinite sequences `id0`, `id1`, `infin5`. The problem, `id2` only requires an integer sequence with finitely many properties.

Starvation-freeness and pattern examples This class is inspired by the *starvation-freeness* property. For instance, an elevator that always services the most recent request runs the risk of starvation of never reaching the first floor if keep coming from floor 2 and 3. Describing that starvation-freeness is not preserved, requires an infinite model. Such properties are well studied in temporal logics and are closely related to automata [35, 46]. Hence, we consider monadic functions from integers (representing time) that must follow some pattern. A simple example is a "flipper" function $f : \mathbb{Z} \to \mathbb{B}$ s.t. $f(x + 1) = \neg f(x)$. We also remark that monadic predicates, as a special case, are also well studied [47]. The problems `infin3` and `infin4` correspond to starvation-freeness and `infin_flip` and `infin_pattern` to the flipper and pattern examples respectively.

Hammer Systems Hammer tools [16] provide the strongest most general automation for proof assistants [28] today. They combine AI with translation to automated theorem proving tools including SMT solvers [15]. As such they are a large source of SMT problems. In principle, the generated SMT problems originate from proof assistant conjectures that the user deemed provable, in practice there are several reasons why the problems are often satisfiable. First, conjectures initially stated by users are sometimes initially incomplete. Second, the AI part of a hammer might not select all the relevant facts. Third, and most importantly, a large part of the generated SMT problems actually come from proof minimisation: in order to reconstruct the simplest proof assistant proof, heuristically parts of the problem are pruned and checked for satisfiability.

The current SMTLIB benchmarks version 2 already includes a large number of problems originating from Sledgehammer [15]. Among the 3462 problems in the theory of UFLIA (linear integer linear arithmetic with uninterpreted functions), 1521 are known to be unsatisfiable and only 14 are known to be satisfiable. We believe that many of the remaining 1927 are satisfiable, but showing this requires infinite models. For several of these problems our heuristic presented in Section 5 is in fact able to find such infinite models. As there are really many possible SMTLIB problems originating from hammers, we select 209 problems with the status `unknown` from Sledgehammer/Hoare.

4 Interpreting Functions using SyGuS

We will experiment with two approaches for synthesising models. The first one uses SyGuS for this task. In our preliminary experiments we only looked at integers (i.e. no infinite datastructures etc.). In particular, we translated the SMT problems to SyGuS problems by keeping all constraints/assertions and changing SMT `declare-fun` commands to SyGuS `synth-fun` commands. Thus tasking the synthesisers to construct functions such that the assertions are valid. As already discussed, the two tasks are very close. This is illustrated by the following example. Note that the `synth-fun` command admits a syntactic constraint. However, by omitting this, we allow the function `f` to form any expression that is permissible within the `UFLIA` theory.

```
(set-logic UFLIA)
(declare-fun f (Int) Int)
(assert (forall ((x Int)) (< x (f x))))
(check-sat)
(get-model)
```

(a) Sample SMT problem

```
(set-logic UFLIA)
(synth-fun f ((x Int)) Int)
(constraint (forall ((x Int)) (< x (f x))))
(check-synth)
```

(b) Corresponding SyGuS translation

5 A Heuristic for Interpreting Sorts and Functions

The direct use of SyGuS presented in the previous section is already useful, but it is not robust enough. In this section we improve on this by proposing and implementing[3] a heuristic method.

Suppose the SMT problem consists of n uninterpreted sorts $\alpha_1, \ldots, \alpha_n$, m uninterpreted functions $f_1, \ldots, f_m$ and k assertions $\varphi_1, \ldots, \varphi_k$. We assume each function f_j has a type $\sigma_1 \times \cdots \times \sigma_l \to \tau$ where each of $\sigma_1, \ldots, \sigma_l$ and τ is either Bool (the two element sort with which SMT solvers are familiar), Int (the infinite sort of integers with which SMT solvers are familiar) or one of $\alpha_1, \ldots, \alpha_k$. In order to demonstrate satisfiability, we must interpret the sorts and function symbols in a way that the assertions are all true. We will do this by interpreting the sorts and functions (possibly with some remaining constants to be interpreted) and make use of the SMT solver `cvc5` to test satisfiability of the problem obtained by translating according to the interpretation. In all our experiments we call `cvc5` with a timeout of a second. All the results reported below are with a total timeout of 1 hour for the procedure.

We will interpret the sorts and functions in two separate phases. The first phase will interpret each sort α_i as either the singleton sort Unit, the two element sort Bool or the infinite sort Int. The second phase will interpret each function f_j as a function expressible in the basic grammar given below. Given such an interpretation (or partial interpretation), we can translate the interpreted problem into a new SMT problem. If the new SMT problem is satisfiable, then we know the original SMT problem is satisfiable. One slight complication is that `cvc5` does not have a Unit type. This can be dealt with by erasing all references to the Unit type via the following recipe:

1. If f_j has type $\sigma_1 \times \cdots \times \sigma_l \to \alpha_i$ where α_i is interpreted as Unit, then f_j is deleted from the problem.

2. For other f_j of type $\sigma_1 \times \cdots \times \sigma_l \to \tau$, we delete each argument position i' where $\sigma_{i'}$ is α_i where α_i is interpreted as Unit.

3. In assertions φ_j every quantifier $\forall x : \alpha_i$ or $\exists x : \alpha_i$ is deleted if α_i is interpreted as Unit, and every equation over the unit type is replaced by $\top$.

5.1 Phase 1: Interpreting sorts

A *partial sort interpretation* Φ is a partial function from $\{\alpha_1, \ldots, \alpha_n\}$ to $\{\mathsf{Unit}, \mathsf{Bool}, \mathsf{Int}\}$. Before determining a partial sort intepretation, we scan the assertions in order to determine if a sort must contain more than one element and for clues that a sort should be infinite.

1. For each φ_j we check if there is a disequation over terms of sort α_i. In this case we record that α_i should not be Unit.

[3] The code is available at http://grid01.ciirc.cvut.cz/~chad/smtintfuncsynth-0.3.tgz.

2. For each φ_j we check for equations with clues that a sort α_i should be infinite. Such a "clue" is a subformula of the form
$$\forall x : \mathsf{Int}.\psi_1 \wedge \ldots \wedge \psi_l \rightarrow g(\ldots, h(\ldots, x, \ldots), \ldots) = x$$
where $h(\ldots, x, \ldots)$ has sort α_i and in every ψ_k either x is not free or ψ_k has the form $x \leq t$ or $x < t$ where x is not free in t. If such a clue is a positive subformula of φ_j, then we commit α_i to be Int.

For each $i \in \{0, \ldots, n\}$, we define Φ_i interpreting the sorts $\alpha_1, \ldots, \alpha_i$, iterating over i. Clearly Φ_0 is the empty partial function. Assuming we have Φ_i, we define Φ_{i+1} by extending the function to assign $\Phi_{i+1}(\alpha_{i+1}) \in \{\mathsf{Unit}, \mathsf{Bool}, \mathsf{Int}\}$. If there is a clue that α_i should be infinite, then we simply choose $\Phi_{i+1}(\alpha_{i+1}) = \mathsf{Int}$. Otherwise, we have two cases: either we know from an assertion that α_{i+1} should not be Unit or we do not. Assume we know it should not be Unit. In that case we let Ψ be the partial function extending Φ_i with $\Psi(\alpha_{i+1}) = \mathsf{Bool}$ and call `cvc5` on the problem interpreted via Ψ. If `cvc5` reports the problem is satisfiable, then the original problem is already satisfiable and we are done. If `cvc5` reports the problem is unsatisfiable, then we assign $\Phi_{i+1}(\alpha_{i+1}) = \mathsf{Int}$. Otherwise, we try calling `cvc5` with two different options: first, with the `--cegqi-all` option; second, with the `--enum-inst` option. We report satisfiability or unsatisfiability if either of these calls succeed. All `cvc5` calls are done with a 1 second timeout which guarantees the termination of the procedure. Otherwise, we assign $\Phi_{i+1}(\alpha_{i+1}) = \mathsf{Bool}$. Next, assume we do not know α_{i+1} should not be Unit. We follow a similar procedure of trying partial interpretation sending α_{i+1} to Unit and using this if the translated problem is not reported as unsatisfiable. Otherwise we try a partial interpretation sending α_{i+1} to Bool and use this if the translated problem is not reported as unsatisfiable (by one of three calls to `cvc5`), and finally sending α_{i+1} to Int otherwise.

At the end of this iteration, we have Φ_n interpreting each sort α_i as one of the three sorts Unit, Bool or Int.

5.2 Phase 2: Interpreting functions

After the first phase, we may simply assume there are no uninterpreted sorts. Furthermore, by erasing references to the Unit type, we can assume all sorts are Bool or Int. What remains is a problem with uninterpreted functions $f_1, \ldots, f_m$ where each function takes booleans and integers as arguments and returns either a boolean or an integer. We restrict our consideration to functions expressible as terms in the following grammar:

$$\begin{array}{rcl} \text{ints}(t) & ::= & x \mid c \mid \mathsf{halve}\ t \mid \mathsf{double}\ t \mid c + t \mid c - t \mid \mathsf{ite}\ p\ t\ t \\ \text{bools}(p) & ::= & \top \mid \bot \mid b \mid t \leq 0 \mid \mathsf{even}\ t \mid \neg p \end{array}$$

In the grammar x is a single variable of type Int and will in practice correspond to one argument of the uninterpreted function. (That is, we never consider functions that depend on more than one input.) The constants c and b are intended as schemes ranging over constants of type Int and Bool. After generating a t or p in the grammar above, we make it specific by creating fresh constants $c_1, c_2, \ldots$ and $b_1, b_2, \ldots$ whose values the SMT solver can determine. The built-in function halve, double, $+$, $-$, ite (if-then-else), $\leq$ and even have straightforward counterparts in the SMT theory of linear arithmetic.

We generate t and p via the grammar up to a certain depth (heuristically selected depth 7), using a heuristic filter to avoid generating all possibilities. Various syntactic features are used to rank the possibilities for t and p at a certain depth and we only keep the top 60 at each

Problem	cvc5-SMT	Z3	SyGuS (cvc5)	Heuristic	Solution (SyGuS)
`id0`	✗	✗	✓	✓	`(def f ((x Int)) Int (+ 1 x))`
`id1`	✗	✓	✓	✓	`(def f ((x Int)) Int x)`
`id2`	✓	✓	✓	✓	Long case split for each assertion.
`infin0`	✗	✗	✗	✓	
`infin1`	✗	✗	✗	✗	

`infin2`	✗	✗	✓	✓	`(def f ((x Int)) Int x)`
					`(def sk ((x Int)) Int (+ x 1))`

`infin3`	✗	✗	✗	✓	
`infin4`	✗	✗	✗	✓	
`infin5`	✗	✓	✗	✓	
`infin_flip`	✗	✗	✗	✓	
`infin_pattern`	✗	✗	✗	✓	
`1035171`	✗	✗	✗	✓	
`1046464`	✗	✗	✗	✓	
`932605`	✗	✗	✗	✓	
`993842`	✗	✗	✗	✓	
`935843`	✗	✗	✗	✓	

Table 1: Results of running SMT solvers (cvc5 and z3) on the SMT problems, cvc5-SyGuS on the corresponding SyGuS problems, and the proposed heuristic. In the last column we show the solutions to the SyGuS problems (where `def` is short for `define-fun` and the solution to `id2` was omitted for space).

depth. Without these heuristic filters, too many candidates would need to be considered and the procedure would not have a chance to succeed on the examples. Once this is done, we have a collection of candidates $t_1, \ldots, t_N$ and $p_1, \ldots, p_M$. For each f_j of type $\sigma_1 \times \ldots \times \sigma_l \to \mathsf{Int}$, each i with $\sigma_i = \mathsf{Int}$, and each $k \in \{1, \ldots, N\}$ we consider interpreting f_j as t_k (treating x as argument i of f_j and ignoring other arguments, and splitting the constants c and b into distinct constants). Likewise for each f_j of type $\sigma_1 \times \ldots \times \sigma_l \to \mathsf{Bool}$, each i with $\sigma_i = \mathsf{Int}$, and each $k \in \{1, \ldots, M\}$ we consider interpreting f_j as p_k (again treating x as argument i of f_j and ignoring other arguments and making constant occurrences distinct).

In most cases for the 209 Sledgehammer/Hoare problems, this will lead to a large number of more specific SMT problems with no uninterpreted sorts and where the only “uninterpreted functions” are actually constants (i.e., functions with no arguments) of type Bool or Int. We call `cvc5` on each of these possibilities, reporting satisfiability on the original problem if `cvc5` reports satisfiability on one of the more specific interpreted SMT problems. Each `cvc5` call is performed with a 1 second timeout, establishing termination of the procedure.

6 Results

In our experiments we ran cvc5 (in SMT mode) [10] as well as Z3 [22] on a set of SMT problems. Subsequently, we also ran cvc5 in SyGuS mode on the equivalent SyGuS problems. Finally, we evaluated the proposed heuristic method on them.

The results are presented in Table 1. There are several problems where employing a solver specifically geared towards synthesis rather than SMT outperforms the corresponding SMT solvers. For example, `id0` and `infin2` could not be solved by either the SMT solvers directly, but SyGuS mode can find a solution. Similarly, for `id1` and `id2`, cvc5 does not find a solution directly, but manages to do so with SyGuS.

The proposed heuristic performs even better, quickly solving the problems with relatively simple solutions. For example, suppose there is one uninterpreted function $f : \mathsf{Int} \to \mathsf{Int}$ that should satisfy $\forall x.x < f(x)$. Interpreting $f(x)$ as $c_0 + x$ gives a solution that `cvc5` can easily verify as satisfiable (presumably taking c_0 to be 1).

The example `infin0` requires us to interpret a sort α, a function $f : \alpha \to \alpha$ and a constant $out : \alpha$ such that f is injective but out is not in the image of f. The proposed procedure can determine that α cannot be Unit or Bool and so interprets α as Int. Once this choice is made, an f of the form $f(x) = c_0 - (c_1 + \mathsf{double}(x))$ gives a solution, where `cvc5` is used to determine values for out, c_0 and c_1.

The example `infin1` requires us to interpret a sort α, a binary relation $R : \alpha \times \alpha \to \mathsf{Bool}$ and a unary function $sk : \alpha \to \alpha$ such that R is transitive and irreflexive and $R\ x\ (sk\ x)$ holds for all x. We can again rule out Unit and Bool for α, leaving us to interpret α as Int. In this case, however, we cannot find an appropriate value of R, as we only consider functions that depend on at most one of their arguments. It should be clear that an appropriate R (e.g., $<$) would need to depend on both arguments.

In the next two example problems, `infin3` and `infin4`, satisfiability means starvation-freeness. Suppose we have one uninterpreted sort α and one function $f : \mathsf{Int} \to \alpha$ and three constants $a, b, c : \alpha$. Suppose these should satisfy the assertions that a, b and c are distinct, $f(x)$ is always one of a, b or c, $f(x) \neq f(x-1)$ and $\forall x.f(x) \neq a$. We can use `cvc5` to easily rule out interpreting α as Unit or Bool, leaving us with α interpreted as Int. Interpreting $f(x)$ as $\mathsf{ite}\ (\mathsf{even}\ x)\ c_1\ c_2$ is sufficient for `cvc5` to determine satisfiability (leaving the SMT solver to determine appropriate values for a, b, c, c_1 and c_2).

More impressive examples that can be solved by the method are five UFLIA problems classified under `sledgehammer/Hoare` originating from sledgehammer (as discussed in Sec. 3): `smtlib.1035171.smt2`, `smtlib.1046464.smt2`, `smtlib.932605.smt2`, `smtlib.993842.smt2`, and `z3.935843.smt2`. Each of these problems is given status “unknown” in SMTLIB at the moment, but we can now classify as satisfiable. The five problems have 10–17 uninterpreted sorts and 20–29 uninterpreted functions. For each of the problems, the procedure interprets one sort as Int, 2–4 sorts as Bool and the rest as Unit. The other 204 problems still remain unknown.

7 Related Work

A large body of research exists on *finite* model finding with numerous techniques anchored in SAT [21, 30, 38], constraint programming [9, 32, 48, 49], SMT [41, 42], and dedicated procedures [17]. The calculated models find a bevy of applications in automated reasoning and computational algebra as counterexamples to incorrect conjectures [14] or for mathematicians interested in particular mathematical structures [23]. Finite models have also been used as a semantic feature in machine-learning guided premise selection [45]. Claessen and Lillieström focus on a bridge between finite and infinite models by providing techniques that attempt to determine a given first-order logic formula has no finite model [20].

When it comes to infinite models, we find much fewer results despite the fact that many practical theories only admit infinite models. Strictly speaking, any SMT solver working with integers is able to produce an infinite model. However, as demonstrated in our experimental validation, state-of-the-art SAT solvers do not go very far in the presence of quantifiers and uninterpreted functions. Similarly, uninterpreted sorts pose a difficult problem.

Model-guided quantifier instantiation [24] in SMT instantiates quantifiers so that they contradict the current, candidate, model. But since the candidate model is constructed based on the ground (partial) instantiation of the formula, it is not clear in general how to extrapolate to complex infinite models.

An area related to the problem at hand is *inductive logic programming*, which enables inventing complex predicates based on positive and negative examples [25, 37]. Also, finite automata are good representatives for infinite behaviors [46]. One might envision collaboration between SMT and these areas. Arif *et al.* [8] explores one such connection by learning temporary logic formulas from finite traces.

Peltier explores techniques for the construction of infinite models in the context of saturation-based theorem proving [34, 34]. Indeed, under ordered resolution, nontrivial certifiable problems may saturate, even though saturation occurs rarely in practice. To our best knowledge, there is no mainstream theorem prover implementing the proposed techniques.

8 Conclusion and Future Work

We propose infinite model finding as a fruitful research domain in satisfiability modulo theories (SMT). Models of theories are undoubtedly interesting, which is witnessed by a large body of research on *finite* model finding. Infinite models, however, are underresearched and may even seem too much of a daunting task. One approach worth mentioning is the generation of models implemented in AGES [26]. Both papers show that infinite model finding is not at all a hopeless task. On the contrary, many opportunities exist to advance the field.

On the one hand, our experiments show that state-of-the-art SMT solvers only enable producing trivial infinite models. On the other hand, SMT solvers *could* expand their capabilities by incorporating techniques from program and function synthesis. That alone, might not be sufficient since current synthesizers do not handle well problems with multiple unknown (uninterpreted) functions and unknown sorts. As a partial answer to this issue, we propose a heuristic approach to instantiate by sorts and functions from the fixed set. These are not instantiated blindly but make use of specific properties mined from the formula. Our heuristic approach is able to solve several problems in the SMT-LIB currently labeled as “unknown”. This gives us an exciting avenue for further research by considering a larger set of functions and properties. Further capabilities could be obtained by integrating with techniques from inductive logic programming and reasearch on temporal logics.

Acknowledgment

The results were supported by the Ministry of Education, Youth and Sports within the dedicated program ERC CZ under the project *POSTMAN* no. LL1902 as well as the ERC starting grant no. 714034 *SMART* and Cost action CA20111 EuroProofNet. This scientific article is part of the *RICAIP* project that has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 857306.

References

[1] A. Abate, C. David, P. Kesseli, D. Kroening, and E. Polgreen. Counterexample guided inductive synthesis modulo theories. In *CAV (1)*, volume 10981 of *LNCS*, pages 270–288. Springer, 2018.

[2] A. Albarghouthi, S. Gulwani, and Z. Kincaid. Recursive program synthesis. In *CAV*, volume 8044 of *LNCS*, pages 934–950. Springer, 2013.

[3] R. Alur, R. Bodík, E. Dallal, D. Fisman, P. Garg, G. Juniwal, H. Kress-Gazit, P. Madhusudan, M. M. K. Martin, M. Raghothaman, S. Saha, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Dependable Software Systems Engineering*, volume 40 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 1–25. IOS Press, 2015.

[4] R. Alur, A. Radhakrishna, and A. Udupa. Scaling enumerative program synthesis via divide and conquer. In *TACAS (1)*, volume 10205 of *LNCS*, pages 319–336, 2017.

[5] R. Alur, R. Singh, D. Fisman, and A. Solar-Lezama. Search-based program synthesis. *Commun. ACM*, 61(12):84–93, 2018.

[6] R. Alur, P. Černý, and A. Radhakrishna. Synthesis through unification. In *CAV (2)*, volume 9207 of *LNCS*, pages 163–179. Springer, 2015.

[7] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Kluwer Academic Publishers, 2nd edition, 2002.

[8] M. F. Arif, D. Larraz, M. Echeverria, A. Reynolds, O. Chowdhury, and C. Tinelli. SYSLITE: syntax-guided synthesis of PLTL formulas from finite traces. In *2020 Formal Methods in Computer Aided Design, FMCAD 2020*, pages 93–103. IEEE, 2020.

[9] G. Audemard, B. Benhamou, and L. Henocque. Predicting and detecting symmetries in FOL finite model search. *J. Autom. Reason.*, 36(3):177–212, 2006.

[10] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar. cvc5: A versatile and industrial-strength SMT solver. In D. Fisman and G. Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022*. Springer, 2022.

[11] C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[12] C. Barrett and C. Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018.

[13] J. Biendarra, J. C. Blanchette, A. Bouzy, M. Desharnais, M. Fleury, J. Hölzl, O. Kunčar, A. Lochbihler, F. Meier, L. Panny, A. Popescu, C. Sternagel, R. Thiemann, and D. Traytel. Foundational (co)datatypes and (co)recursion for higher-order logic. In C. Dixon and M. Finger, editors, *Frontiers of Combining Systems - 11th International Symposium, FroCoS 2017*, volume 10483 of *LNCS*, pages 3–21. Springer, 2017.

[14] J. C. Blanchette. Nitpick: A counterexample generator for Isabelle/HOL based on the relational model finder Kodkod. In A. Voronkov, G. Sutcliffe, M. Baaz, and C. G. Fermüller, editors, *Short papers for 17th International Conference on Logic for Programming, Artificial intelligence, and Reasoning, LPAR-17-short*, volume 13 of *EPiC Series in Computing*, pages 20–25. EasyChair, 2010.

[15] J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending Sledgehammer with SMT solvers. *J. Autom. Reason.*, 51(1):109–128, 2013.

[16] J. C. Blanchette, C. Kaliszyk, L. C. Paulson, and J. Urban. Hammering towards QED. *J. Formalized Reasoning*, 9(1):101–148, 2016.

[17] S. Borgwardt and B. Morawska. Finding finite Herbrand models. In N. S. Bjørner and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 18th International Conference, LPAR 2012*, volume 7180 of *LNCS*, pages 138–152. Springer, 2012.

[18] J. Cambronero, S. Gulwani, V. Le, D. Perelman, A. Radhakrishna, C. Simon, and A. Tiwari. FlashFill++: Scaling programming by example by cutting to the chase. *Proc. ACM Program. Lang.*, 7(POPL):952–981, 2023.

[19] B. Caulfield, M. N. Rabe, S. A. Seshia, and S. Tripakis. What's decidable about syntax-guided

synthesis? *CoRR*, abs/1510.08393, 2015.

[20] K. Claessen and A. Lillieström. Automated inference of finite unsatisfiability. *J. Autom. Reason.*, 47(2):111–132, 2011.

[21] K. Claessen and N. Sörensson. New techniques that improve MACE-style finite model finding. In *Proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications*, 2003.

[22] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[23] A. Distler, M. Shah, and V. Sorge. Enumeration of AG-groupoids. In J. H. Davenport, W. M. Farmer, J. Urban, and F. Rabe, editors, *Intelligent Computer Mathematics, CICM 2011*, volume 6824 of *LNCS*, pages 1–14. Springer, 2011.

[24] Y. Ge and L. M. de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In A. Bouajjani and O. Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009*, volume 5643 of *LNCS*, pages 306–320. Springer, 2009.

[25] S. Gulwani, J. Hernández-Orallo, E. Kitzelmann, S. H. Muggleton, U. Schmid, and B. G. Zorn. Inductive programming meets the real world. *Commun. ACM*, 58(11):90–99, 2015.

[26] R. Gutiérrez and S. Lucas. Automatic generation of logical models with AGES. In P. Fontaine, editor, *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction*, volume 11716 of *LNCS*, pages 287–299. Springer, 2019.

[27] J. Harrison. HOL Light: An overview. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *LNCS*, pages 60–66. Springer, 2009.

[28] J. Harrison, J. Urban, and F. Wiedijk. History of interactive theorem proving. In J. H. Siekmann, editor, *Computational Logic*, volume 9 of *Handbook of the History of Logic*, pages 135–214. Elsevier, 2014.

[29] K. Huang, X. Qiu, P. Shen, and Y. Wang. Reconciling enumerative and deductive program synthesis. In *PLDI*, pages 1159–1174. ACM, 2020.

[30] M. Janota and M. Suda. Towards smarter MACE-style model finders. In G. Barthe, G. Sutcliffe, and M. Veanes, editors, *22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR 2018*, volume 57 of *EPiC Series in Computing*, pages 454–470. EasyChair, 2018.

[31] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In *PLDI*, pages 316–329. ACM, 2010.

[32] W. McCune. Mace4 reference manual and guide. *CoRR*, cs.SC/0310055, 2003.

[33] C. Okasaki. *Purely functional data structures.* Cambridge University Press, 1999.

[34] N. Peltier. Model building with ordered resolution: extracting models from saturated clause sets. *J. Symb. Comput.*, 36(1-2):5–48, 2003.

[35] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977.

[36] O. Polozov and S. Gulwani. FlashMeta: a framework for inductive program synthesis. In *OOPSLA*, pages 107–126. ACM, 2015.

[37] S. J. Purgał, D. M. Cerna, and C. Kaliszyk. Learning higher-order logic programs from failures. In L. D. Raedt, editor, *Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022*, pages 2726–2733, 2022.

[38] G. Reger, M. Riener, and M. Suda. Symmetry avoidance in MACE-style finite model finding. In A. Herzig and A. Popescu, editors, *Frontiers of Combining Systems - 12th International Sympo-*

sium, FroCoS 2019, volume 11715 of *LNCS*, pages 3–21. Springer, 2019.

[39] A. Reynolds, H. Barbosa, A. Nötzli, C. W. Barrett, and C. Tinelli. cvc4sy: Smart and fast term enumeration for syntax-guided synthesis. In *CAV (2)*, volume 11562 of *LNCS*, pages 74–83. Springer, 2019.

[40] A. Reynolds, V. Kuncak, C. Tinelli, C. W. Barrett, and M. Deters. Refutation-based synthesis in SMT. *Formal Methods Syst. Des.*, 55(2):73–102, 2019.

[41] A. Reynolds, C. Tinelli, A. Goel, and S. Krstić. Finite model finding in SMT. In *Computer Aided Verification - 25th International Conference, CAV*, pages 640–655, 2013.

[42] A. Reynolds, C. Tinelli, A. Goel, S. Krstić, M. Deters, and C. Barrett. Quantifier instantiation techniques for finite model finding in SMT. In *24th International Conference on Automated Deduction, CADE 2013*, pages 377–391, 2013.

[43] A. Solar-Lezama. Program sketching. *Int. J. Softw. Tools Technol. Transf.*, 15(5-6):475–495, 2013.

[44] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017.

[45] J. Urban, G. Sutcliffe, P. Pudlák, and J. Vyskočil. MaLARea SG1 – machine learner for automated reasoning with semantic guidance. In *International Joint Conference on Automated Reasoning (IJCAR)*, pages 441–456, 2008.

[46] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In F. Moller and G. M. Birtwistle, editors, *Logics for Concurrency - Structure versus Automata (8th Banff Higher Order Workshop)*, volume 1043 of *LNCS*, pages 238–266. Springer, 1995.

[47] M. Veanes, N. S. Bjørner, L. Nachmanson, and S. Bereg. Effectively monadic predicates. In K. L. McMillan, A. Middeldorp, G. Sutcliffe, and A. Voronkov, editors, *19th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR 2013*, volume 26 of *EPiC Series in Computing*, pages 97–103. EasyChair, 2013.

[48] J. Zhang. Constructing finite algebras with FALCON. *Journal of Automated Reasoning*, 17:1–22, 08 1996.

[49] J. Zhang and H. Zhang. SEM: a system for enumerating models. In *IJCAI*, pages 298–303, 1995.