Experiments on Infinite Model Finding in SMT Solving

Julian Parsert¹, Chad E. Brown², Mikoláš Janota², and Cezary Kaliszyk³

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 \( (T,F,\phi,G) \) where \( T \) is a background theory, \( F \) is a second-order variable, \( \phi \) 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 \( (LIA,\text{max2},\phi,G) \) where \( \phi \) is

\[
\text{max2}(x,y) \geq x \land \text{max2}(x,y) \geq y \land (\text{max2}(x,y) = x \lor \text{max2}(x,y) = y)
\]

and \( G \) is

\[
I \rightarrow x \mid y \mid 0 \mid 1 \mid I + I \mid I - I \mid (\text{ite}^2 BI)
\]

\[
B \rightarrow B \land B \mid B \lor B \mid \neg B \mid I = I \mid I < I \mid I > I.
\]

This describes a SyGuS problem in linear integer arithmetic (LIA) where specification \( \phi \) describes the maximum of the two arguments to \( \text{max2} \). 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:

```lisp
(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.
Experiments on Infinite Model Finding in SMT

Parsert, Brown, Janota, and Kaliszyk

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

The following expression would be a candidate 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.
Experiments on Infinite Model Finding in SMT

(a) Sample SMT problem

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

(b) Corresponding SyGuS translation

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

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 \(\sigma_1 \times \cdots \times \sigma_l \rightarrow \tau\) where each of \(\sigma_1, \ldots, \sigma_l\) and \(\tau\) is either \(\text{Bool}\) (the two element sort with which SMT solvers are familiar), \(\text{Int}\) (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 \(\text{cvc5}\) to test satisfiability of the problem obtained by translating according to the interpretation. In all our experiments we call \(\text{cvc5}\) 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 \(\alpha_i\) as either the singleton sort \(\text{Unit}\), the two element sort \(\text{Bool}\) or the infinite sort \(\text{Int}\). 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 \(\text{cvc5}\) does not have a \(\text{Unit}\) type. This can be dealt with by erasing all references to the \(\text{Unit}\) type via the following recipe:

1. If \(f_j\) has type \(\sigma_1 \times \cdots \times \sigma_l \rightarrow \alpha_i\) where \(\alpha_i\) is interpreted as \(\text{Unit}\), then \(f_j\) is deleted from the problem.
2. For other \(f_j\) of type \(\sigma_1 \times \cdots \times \sigma_l \rightarrow \tau\), we delete each argument position \(i'\) where \(\sigma_{i'}\) is \(\alpha_i\) where \(\alpha_i\) is interpreted as \(\text{Unit}\).
3. In assertions \(\varphi_j\) every quantifier \(\forall x : \alpha_i\) or \(\exists x : \alpha_i\) is deleted if \(\alpha_i\) is interpreted as \(\text{Unit}\), and every equation over the unit type is replaced by \(\top\).

5.1 Phase 1: Interpreting sorts

A partial sort interpretation \(\Phi\) is a partial function from \(\{\alpha_1, \ldots, \alpha_n\}\) to \(\{\text{Unit}, \text{Bool}, \text{Int}\}\). Before determining a partial sort interpretation, 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 \(\varphi_j\) we check if there is a disequation over terms of sort \(\alpha_i\). In this case we record that \(\alpha_i\) should not be \(\text{Unit}\).

\(^3\)The code is available at \(\text{http://grid01.ciirc.cvut.cz/~chad/smtintfuncsynth-0.3.tgz}\).
2. For each \( \varphi_j \) we check for equations with clues that a sort \( \alpha_i \) should be infinite. Such a “clue” is a subformula of the form

\[
\forall x : \text{Int}. \psi_1 \land \ldots \land \psi_l \rightarrow g(\ldots, h(\ldots, x, \ldots), \ldots) = x
\]

where \( h(\ldots, x, \ldots) \) has sort \( \alpha_i \) and in every \( \psi_k \) either \( x \) is not free or \( \psi_k \) 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 \( \varphi_j \), then we commit \( \alpha_i \) to be \text{Int}.

For each \( i \in \{0, \ldots, n\} \), we define \( \Phi_i \) interpreting the sorts \( \alpha_1, \ldots, \alpha_i \), iterating over \( i \). Clearly \( \Phi_0 \) is the empty partial function. Assuming we have \( \Phi_i \), we define \( \Phi_{i+1} \) by extending the function to assign \( \Phi_{i+1}(\alpha_{i+1}) \in \{\text{Unit}, \text{Bool}, \text{Int}\} \). If there is a clue that \( \alpha_i \) should be infinite, then we simply choose \( \Phi_{i+1}(\alpha_{i+1}) = \text{Int} \). Otherwise, we have two cases: either we know from an assertion that \( \alpha_{i+1} \) should not be \text{Unit} or we do not. Assume we know it should not be \text{Unit}. In that case we let \( \Psi \) be the partial function extending \( \Phi_i \) with \( \Psi(\alpha_{i+1}) = \text{Bool} \) and call \text{cvc5} on the problem interpreted via \( \Psi \). If \text{cvc5} reports the problem is satisfiable, then the original problem is already satisfiable and we are done. If \text{cvc5} reports the problem is unsatisfiable, then we assign \( \Phi_{i+1}(\alpha_{i+1}) = \text{Int} \). Otherwise, we try calling \text{cvc5} with two different options: first, with the \text{-cegqi-all} option; second, with the \text{-enum-inst} option. We report satisfiability or unsatisfiability if either of these calls succeed. All \text{cvc5} calls are done with a 1 second timeout which guarantees the termination of the procedure. Otherwise, we assign \( \Phi_{i+1}(\alpha_{i+1}) = \text{Bool} \).

Next, assume we do not know \( \alpha_{i+1} \) should not be \text{Unit}. We follow a similar procedure of trying partial interpretation sending \( \alpha_{i+1} \) to \text{Unit} and using this if the translated problem is not reported as unsatisfiable. Otherwise we try a partial interpretation sending \( \alpha_{i+1} \) to \text{Bool} and use this if the translated problem is not reported as unsatisfiable (by one of three calls to \text{cvc5}), and finally sending \( \alpha_{i+1} \) to \text{Int} otherwise.

At the end of this iteration, we have \( \Phi_n \) interpreting each sort \( \alpha_i \) as one of the three sorts \text{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 \text{Unit} type, we can assume all sorts are \text{Bool} or \text{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:

\[
\begin{align*}
t & ::= x \mid c \mid \text{halve } t \mid \text{double } t \mid c + t \mid c - t \mid \text{ite } p \ t \ t \\
p & ::= \top \mid \bot \mid b \mid t \leq 0 \mid \text{even } t \mid \neg p
\end{align*}
\]

In the grammar \( x \) is a single variable of type \text{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 \text{Int} and \text{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 \text{halve}, \text{double}, \text{+, -, }\text{ite (if-then-else), }\leq \text{ 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
Experiments on Infinite Model Finding in SMT

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).

<table>
<thead>
<tr>
<th>Problem</th>
<th>cvc5-SMT</th>
<th>Z3</th>
<th>SyGuS (cvc5)</th>
<th>Heuristic</th>
<th>Solution (SyGuS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>id0</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
<td>(def f ((x Int)) Int (+ 1 x))</td>
</tr>
<tr>
<td>id1</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>(def f ((x Int)) Int x)</td>
</tr>
<tr>
<td>id2</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>(def sk ((x Int)) Int (+ x 1))</td>
</tr>
<tr>
<td>infin0</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>infin1</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
</tr>
<tr>
<td>infin2</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>infin3</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>infin4</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>infin5</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>infin_flip</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>infin_pattern</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>1035171</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>1046464</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>932605</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>993842</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>935843</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td></td>
</tr>
</tbody>
</table>

Table 6

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, \texttt{id0} and \texttt{infin2} could not be solved by either the SMT solvers directly, but SyGuS mode can find a solution. Similarly, for \texttt{id1} and \texttt{id2}, \texttt{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 \( f : \text{Int} \rightarrow \text{Int} \) that should satisfy \( \forall x. x < f(x) \). Interpreting \( f(x) \) as \( c_0 + x \) gives a solution that \texttt{cvc5} can easily verify as satisfiable (presumably taking \( c_0 \) to be 1).

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

The example \texttt{infin1} requires us to interpret a sort \( \alpha \), a binary relation \( R : \alpha \times \alpha \rightarrow \text{Bool} \) and a unary function \( \text{sk} : \alpha \rightarrow \alpha \) such that \( R \) is transitive and irreflexive and \( R x (\text{sk} x) \) holds for all \( x \). We can again rule out \text{Unit} and \text{Bool} for \( \alpha \), leaving us to interpret \( \alpha \) as \text{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, \texttt{infin3} and \texttt{infin4}, satisfiability means starvation-freeness. Suppose we have one uninterpreted sort \( \alpha \) and one function \( f : \text{Int} \rightarrow \alpha \) 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 \texttt{cvc5} to easily rule out interpreting \( \alpha \) as \text{Unit} or \text{Bool}, leaving us with \( \alpha \) interpreted as \text{Int}. Interpreting \( f(x) \) as \( \text{ite} \) (even \( x \)) \( c_1 \) \( c_2 \) is sufficient for \texttt{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 \textit{sledgehammer/Hoare} originating from sledgehammer (as discussed in Sec. 3): \texttt{smtlib.1035171.smt2}, \texttt{smtlib.1046464.smt2}, \texttt{smtlib.932605.smt2}, \texttt{smtlib.993842.smt2}, and \texttt{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 \text{Int}, 2–4 sorts as \text{Bool} and the rest as \text{Unit}. The other 204 problems still remain unknown.

7 Related Work

A large body of research exists on \textit{finite} model finding with numerous techniques anchored in SAT \cite{21, 30, 38}, constraint programming \cite{9, 32, 48, 49}, SMT \cite{41, 42}, and dedicated procedures \cite{17}. The calculated models find a bevy of applications in automated reasoning and computational algebra as counterexamples to incorrect conjectures \cite{14} or for mathematicians interested in particular mathematical structures \cite{23}. Finite models have also been used as a semantic feature in machine-learning guided premise selection \cite{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 \cite{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

Experiments on Infinite Model Finding in SMT Parsing


[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 Sympos-


