LOIS: an application of SMT solvers*

Eryk Kopczyński and Szymon Toruńczyk

University of Warsaw, Poland
{erykk,szymtor}@mimuw.edu.pl

Abstract

We present an implemented programming language called LOIS, which allows iterating through certain infinite sets, in finite time. We argue that this language offers a new application of SMT solvers to verification of infinite-state systems, by showing that many known algorithms can easily be implemented using LOIS, which in turn invokes SMT solvers for various theories. In many applications, ω-categorical theories with quantifier elimination are of particular interest. Our tests indicate that state-of-the art solvers perform poorly on such theories, as they are outperformed by orders of magnitude by a simple quantifier-elimination procedure.

1 Introduction

A fragment of formal verification is concerned in designing algorithms which test properties of infinite-state systems, such as reachability or verifying whether all runs satisfy a given formula [KF94, KM69, FS01, DHPV09]. There are two components to this process: constructing an algorithm, and proving its correctness and termination. Usually, the algorithm involves specific data structures for representing infinite objects in a finite way. Consequently, the proof of termination and correctness needs to delve into the specifics of these data structures, and is often more complicated than the actual underlying mathematical idea.

This paper and its companion paper [KTb] introduce a programming language called LOIS (*Looping Over Infinite Sets*), which manipulates infinite mathematical objects in a transparent way, by allowing to iterate through infinite sets, in finite time. For instance, the code to the right below can be executed in finite time ($\mathbb{N}$ denotes the set of all naturals); as a result, the set Y evaluates to the set of odd numbers greater than 7.

```
set Y = ∅;
for (x : ℕ)
  if (x>3) Y += 2*x+1;

if (10∈Y)
  cout << "10 is odd";
```

To the best of our knowledge, LOIS is the first imperative programming language which allows to evaluate instructions (possibly, nested) similar to the one above, on infinite sets, in finite time. LOIS is implemented as a prototype library in C++; the above instruction is executable after minor syntactic transformations [KTc]. When executing programs, LOIS invokes SMT (*Satisfiability Modulo Theories*) solvers for testing validity of first-order formulas. The above program amounts to testing the validity of the formula $\exists x.(x > 3) \wedge (2 \cdot x + 1 = 10)$ in $(\mathbb{N}, \cdot, +)$.

We believe that LOIS offers a convenient interface between formal verification, SMT solvers, and abstract mathematical arguments. The main benefit is that no specific data structures need to be employed to manipulate infinite mathematical objects. To illustrate our point, we give another example.

Example 1.1. Below is a simple LOIS program. It constructs an automaton with infinitely many states, which for a given sequence over the alphabet $\Sigma = \mathbb{N} \cup \{\#\}$, computes the maximal sum of an infix not interrupted by #. To the left below, we use pseudocode similar to LOIS syntax.

*This work is supported by Poland's National Science Centre grant 2012/07/B/ST6/01497.

```
set Σ = ℕ∪{'#'};
set Q = ∅;
set I = {(0,0)};
set δ = ∅;

for m in ℕ do
  for n in ℕ do
    Q+= (m,n);

for (m,n) in Q do
  for x in Σ do
    if (x=='#')
      δ+=((m,n),x,(m,0))
    else
      δ+=((m,n),x,
            (max(m,n+x),n+x));

function reach(I,E) {
  set S = ∅;
  set R = I;

  while (R != S) do {
    S = R;
    for (p,q) in E do
      if (p∈R) R += q;
  }
  return R;
}

set E = ∅;

for (p,a,q) in δ do
  E += (p,q);

print(reach(I,E));
```

The statespace Q consists of all pairs (m, n) with $m, n \in \mathbb{N}$. There is one initial state, $(0, 0)$. The transition relation $\delta \subseteq Q \times \Sigma \times Q$ is such that reading # resets the second component of the state, and reading a letter $x \in \mathbb{N}$ increases the second component by x, and accumulates the maximal seen value in the first component. The tuple (Σ, Q, δ, I) is an automaton (without accepting states), with infinite-state space.

To understand LOIS on an intuitive level, imagine that an instruction of the form "`for x in X do I`" creates possibly infinitely many threads indexed by elements $x \in X$, executed in parallel and perfectly synchronously; the thread x executes the instruction `I` with the value of `x` set to x.

We now compute the set of reachable states of the above automaton. The function `reach` takes as argument a possibly infinite set E of directed edges (pairs of vertices), a set of initial vertices I, and computes the reachable set with a fixpoint algorithm. Contrary to `for` loops, `while` loops are executed sequentially. In particular, in a terminating program, they are executed finitely many times.

Finally, we compute the reachable states in the graph of our automaton. The output is $\{(m, n) \mid m \in \mathbb{N}, n \in \mathbb{N}, m \geq n\}$. The while loop iterates three times, with R taking values $\{(0, 0)\}, \{(n, n) \mid n \in \mathbb{N}\}$, and $\{(m, n) \mid m, n \in \mathbb{N}, m \geq n\}$.

Let us specify a set of accepting states, e.g. $F = \{(m, 3) \mid m \in \mathbb{N}\}$, which can be constructed in LOIS, similarly to Q. Now $A = (\Sigma, Q, \delta, I, F)$ is a deterministic, infinite-state automaton accepting those sequences, in which the maximal sum of an infix uninterrupted by # is 3.

```
function minimize(Σ,Q,q₀,F,δ)
{
  set E = ∅;
  for (p,q,a) in Q×Q×Σ do
    E+=((δ(p,a),δ(q,a)),(p,q));
  set S=(F×(Q-F))∪((Q-F)×F);
  set equiv=(Q×Q)-reach(S,E);

  set classes=∅;
  for q in Q do {
    set class = ∅;
    for p in Q do
      if ((p,q)∈equiv)
        class += p;
    classes += class
  }

  return classes
}
```

What is the minimal automaton equivalent to A? To find out, we can try to run the well-known *partition refinement* algorithm on A. Since this only works for deterministic automata, we treat δ as a function $Q \times \Sigma \to Q$ and q_0 is the unique initial state. In the first phase, we compute in the variable `equiv` the equivalence relation which identifies states that recognise the same languages, i.e., $(p, q) \in$ `equiv` iff for all words $w \in \Sigma^*$, reading w from the state p, ends in an accepting state iff it does from the state q. To compute `equiv` we use the function `reach` described earlier. In the second phase, we compute the equivalence classes of the relation `equiv` on Q, which are the states of the minimal automaton; the transitions can be computed similarly. For the automaton A described above, this returns a minimal automaton with 11 states. Note that the same LOIS code as in the `reach` and `minimize` functions can be used for classical, finite automata, as well as for various classes of infinite-state systems. It can be readily converted into a very similar, executable LOIS program (this is done in [KTc]), with no need of auxiliary data structures.

What is and what is not in this paper. Although the syntax of LOIS is simple and its semantics intuitive, a formal treatment requires novel ideas and is deferred to another paper [KTb]. For our purposes here, it suffices to know that each time one of the LOIS instructions `for`, `if`, `while` is executed, an SMT solver is queried for the theory underlying the particular program, as explained in Section 2. The C++ library is described in [KTc]. This paper concentrates on the use of SMT solvers. In particular, we argue

that LOIS provides a new application of SMT solvers to formal verification, using background theories which are not the ones typically considered in the SMT community.

The outline is as follows. LOIS, together with its key underlying data structure, definable sets, are introduced in Section 2. These rely on SMT solvers for various ω-categorical theories, which we describe in Section 3. We show (in Section 4) that they outperform state-of-the-art SMT solvers by orders of magnitude. In Section 5 we give some example applications of LOIS to verification of infinite-state systems. We discuss the related work in Section 6.

2 Definable sets and LOIS

In this section, we define the central notion underlying this paper and the companion paper [KTb], that is, of definable sets. We also briefly introduce LOIS.

Structures. We refer to the literature (e.g. [Hod97]) for structures, sorts, terms, and formulas. All formulas are assumed to be first-order. Throughout the paper, fix an infinite *underlying* logical structure $\mathcal{A}$, which may involve relation or function symbols. For simplicity, we assume that $\mathcal{A}$ has only one sort named $\mathbb{A}$; generalizing to multisorted structures is straightforward. We say that $\mathcal{A}$ has *decidable theory* if there is an algorithm which decides whether a given first-order sentence holds in $\mathcal{A}$. Such an algorithm is called an *SMT solver* for the theory of $\mathcal{A}$. The structures $(\mathbb{N}, =), (\mathbb{Q}, \leq), (\mathbb{R}, \leq), (\mathbb{R}, \leq, +, \cdot), (\mathbb{N}, \leq, +)$ have decidable theories (the last two due to results of Tarski and Presburger), and the structure $(\mathbb{N}, +, \cdot)$ does not, by Gödel's theorem. In this paper the underlying structure $\mathcal{A}$ is always assumed to a have decidable theory.

Definable sets. An *expression* (defined recursively) is either a variable from a fixed infinite set of variables, or a formal finite union of *set-builder expressions*, each of the form

$$\{e \mid a_1 \in \mathbb{A}, \ldots, a_n \in \mathbb{A}, \phi\}, \tag{1}$$

where e is an expression, $a_1, \ldots, a_n$ are (*bound*) variables, and ϕ is a (*guard*) formula over the signature of $\mathcal{A}$ and over the set of variables. Free variables in (1) are those free variables of e and of ϕ which are not among $a_1, \ldots, a_n$. For an expression e with free variables V, any valuation $\mathrm{val} : V \to \mathcal{A}$ defines in the obvious way a value $X = e[\mathrm{val}]$, which is either an element of $\mathcal{A}$ or a set, formally defined by induction on the structure of e. We then say that X is *definable over* $\mathcal{A}$, or *with* underlying structure $\mathcal{A}$. When we want to emphasize those elements of $\mathcal{A}$ that are used in a definition of X, we say that X is S-*definable*, if X is defined using a valuation $\mathrm{val} : V \to S$, for a finite set $S \subseteq \mathcal{A}$.

Example 2.1. Let $\mathcal{Q}$ be the *rationals with order*, with one sort $\mathbb{Q}$ consisting of rational numbers, and a predicate $<$ denoting the usual order on $\mathbb{Q}$. The interval $(1/4, 5/6)$ is $\{1/4, 5/6\}$-definable by the expression $\{x \mid x \in \mathbb{Q}, a < x \wedge x < b\}$ and valuation $a \mapsto 1/4, b \mapsto 5/6$. Also its complement $\mathbb{Q} - (1/4, 5/6)$ is $\{1/4, 5/6\}$-definable. More generally, definable subsets of $\mathbb{Q}$ (over $\mathcal{Q}$) are precisely finite unions of open (possibly half-bounded) intervals and points. The set of all open intervals in $\mathcal{Q}$ is a $\emptyset$-definable set, defined by the expression $\{\{x \mid x \in \mathbb{Q}, a < x \wedge x < b\} \mid a \in \mathbb{Q}, b \in \mathbb{Q}\}$.

Now consider the *ordered field of reals* $\mathcal{R} = (\mathbb{R}, +, \cdot, 0, 1, \leq)$. An example definable subset of $\mathbb{R}^3$ is the half-ball $\{(x, y, z) \mid x \in \mathbb{R}, y \in \mathbb{R}, z \in \mathbb{R}, x > 0 \wedge x^2 + y^2 + z^2 \leq 1\}$. A celebrated result of Tarski characterizes definable subsets of $\mathbb{R}^k$ (over $\mathcal{R}$) as precisely the finite unions of sets defined by systems of equalities and inequalities between k-variate polynomials. The set of all balls in $\mathbb{R}^k$ is also $\emptyset$-definable.

Remark 2.2. In the above example we used a tuple (x, y, z) in an expression. This is syntactic sugar, as tuples can be encoded as finite sets using e.g. Kuratowski pairs. Note that any finite set whose elements are definable is itself definable. We will often use symbols, e.g. #, as expressions, formally represented by $\emptyset$-definable sets, e.g., $\emptyset, \{\emptyset\}$, etc. In LOIS programs, we allow including symbols, tuples, arrays, objects, and other data structures directly in sets.

LOIS. Very roughly, the syntax of LOIS with underlying structure $\mathcal{A}$ extends the syntax of an imperative language (e.g. C++ or Pascal) by:

- the *pseudoparallel* instruction `for x in X`,
- types `set` for representing sets and `elem` for control variables in a `for` loop,
- constants $\emptyset$ and $\mathbb{A}$ (the sort of $\mathcal{A}$) of type `set`,
- set manipulations, such as insertion `X+=x`, and operations $\cap, \cup, -, \times$,
- tests $X = Y, X \in Y, X \subseteq Y, x \in X$ for X, Y of type `set` and x of type `elem`,
- tests $\phi(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are variables of type `elem` and ϕ is a formula using relation and function symbols from $\mathcal{A}$.

See our companion paper [KTb] for the formal semantics[1], and [KTc] for the exhaustive list of constructions available in our implementation. In Section 2 below we discuss how SMT solvers are employed by LOIS.

Definable sets are the central data structure underlying LOIS programs, used to represent elements of type `set`. They are effectively closed under boolean combinations, cartesian products, projections, quotients, etc. In fact, they are closed under any function which can be implemented as a LOIS program, as stated slightly informally below, and proved in our companion paper [KTb].

Theorem 2.3. *Let* `I` *be a LOIS instruction with underlying structure* $\mathcal{A}$ *and let* v *be a valuation which assigns* S*-definable sets to variables appearing in* `I`*, where* $S \subseteq \mathcal{A}$ *is a finite set. Then, executing* `I` *results in a valuation* $[\![I]\!](v)$ *which also assigns* S*-definable sets to the variables appearing in* `I`*. If the theory of* $\mathcal{A}$ *is decidable and the instruction* `I` *has bounded recursion and iteration depth, then the valuation* $[\![I]\!](v)$ *can be effectively computed from* v*.*

Queries generated by LOIS. We briefly describe when LOIS queries an SMT solver. This is part of the formal semantics described in detail in our companion paper [KTb].

A *context* is a finite set of bound variables and formulas. During the execution of a program, LOIS maintains a stack of contexts, modified by the `for`, `if`, `while` instructions. A statement of the form `for x in X do I` is executed as follows. Assume that `X` evaluates to a definable set X, which is internally represented by a union of set-builder expressions $e_1 \cup \cdots \cup e_n$. For each expression e_i in this union, do as follows. Suppose that e_i is of the form $\{f \mid a_1 \in \mathbb{A}, \ldots, a_n \in \mathbb{A}, \phi\}$. Then a context C comprising the bound variables $a_1, \ldots, a_n$ and the formula ϕ is pushed onto the stack. If the union of all contexts currently on the stack is satisfiable, then the instruction I is executed with variable x set to f. Afterwards, the context C is removed from the stack and we move to the next expression e_{i+1}, until all expressions are processed. The instruction `if` ϕ `do I` is equivalent to `for x in` $\{\emptyset \mid \phi\}$ `do I`, and `while` is implemented as a sequential (finite) application of `if` instructions, as usual.

Satisfiability tests. As we see above, during the execution of a program, LOIS performs only a few operations on contexts, which can be described as follows: (`push`) push a context onto the stack, (`pop`) remove the topmost context from the stack, (`check-sat`) check if the union of all contexts on the stack is satisfiable. Conveniently, many SMT solvers – in particular, the solvers conforming to the SMT-LIB standard [BST10] – allow to execute the above three operations (`push`), (`pop`), (`check-sat`), for certain background theories; this is known as *incremental solving*. LOIS can communicate with an external incremental solver, using the SMT-LIB (v. 2) format. Also, for some theories, LOIS can use its internal solver, described in Section 3.

Origin of the formulas. It is perhaps worth expounding on the origin and shape of the formulas appearing in the set-builder expressions during the execution of a LOIS program. Whenever the instruction `X+=x` is executed, a new set-builder expression is appended to the expression defining X; the

[1] In our companion paper [KTb] two languages, LOIS and LOIS_0 are introduced. They have the same syntax, but slightly different semantics. This distinction will not be relevant in this paper. Additionally, there is the tool which is also called LOIS, implemented as a C++ library allowing to execute programs in a syntax similar to that of LOIS (some minor changes are required to embed LOIS into C++).

guard in this expression is the union of those contexts on the stack which appeared after X was declared. Boolean values are represented by formulas. For example, if X is represented by a single set-builder expression of the form $\{f \mid a_1 \in \mathbb{A}, \ldots, a_n \in \mathbb{A}, \phi\}$, then `(X==∅)` evaluates to $\forall a_1 \ldots \forall a_n \neg\phi$. Tests for $\in, =, \subseteq$ are defined mutually recursively – for example, $X \subseteq Y$ is implemented by the code below. For sets X, Y defined by expressions of nesting depth n, the result is a formula with quantifier prefix $\forall^*\exists^* \ldots$ and $2n$ alternations between $\forall$ and $\exists$.

```
function subset(X,Y) {
  set F = ∅;
  for (x : X)
    if !(x ∈ y) F+={∅};
  return (F==∅);
}
```

From time to time, LOIS tries to simplify the formulas appearing in the set-builder expressions, which turns out to be crucial for the performance, since simpler formulas are easier to verify. LOIS also performs basic syntactic transformations, such as removing quantifiers which introduce unused variables. Additionally, LOIS runs its internal solver on the guards when constructing sets, checking whether there are any parts which always turn out to be true or false during the evaluation, and removing them. For relatively simple LOIS programs this simplification algorithm is very effective.

3 Internal solver

LOIS has an internal solver which can handle several ω-categorical theories, in particular, of *homogeneous* structures. These are important for many of the applications sampled in Section 5. In this section, we discuss the algorithm, and test it against state-of-the-art SMT solvers.

ω-categoricity. For a structure $\mathcal{A}$, its *automorphism* is a bijection of $\mathcal{A}$ to itself, which preserves the relations and functions of $\mathcal{A}$. An automorphism π of $\mathcal{A}$ can be applied to a tuple $(a_1, \ldots, a_n)$ of elements of $\mathcal{A}$, yielding as a result the tuple $(\pi(a_1), \ldots, \pi(a_n))$; we say that two tuples $\bar{a}, \bar{b} \in \mathcal{A}^n$ are in the *same orbit* if there is an automorphism which maps $\bar{a}$ to $\bar{b}$. An orbit is an equivalence class of this equivalence relation. A countable structure $\mathcal{A}$ is *ω-categorical* if for every $n \in \mathbb{N}$, the set of tuples $\mathcal{A}^n$ has finitely many orbits.

Example 3.1. Consider the structure $(\mathbb{N}, =)$, whose automorphisms are all bijections of $\mathbb{N}$ to itself. The orbits of $\mathbb{N}^3$ are identified by representatives $(1,1,1), (1,1,2), (1,2,1), (2,1,1), (1,2,3)$. More generally, the orbits of $\mathbb{N}^k$ correspond to partitions of the set $\{1, \ldots, k\}$; in particular, $(\mathbb{N}, =)$ is ω-categorical. The structure $(\mathbb{Q}, \leq)$, whose automorphisms are the increasing bijections of $\mathbb{Q}$, is also ω-categorical. Other ω-categorical structures which arise in formal verification include the infinite random graph [Mac11] and infinite homogeneous trees [BST13].

Homogeneity. Recall that we consider structures with relation and/or function symbols. An *n-generated* structure $\mathcal{B}$ is a structure with an n-tuple of distinguished *generators* from which every other element in $\mathcal{B}$ can be obtained using function symbols. An *isomorphism* of two n-generated structures $\mathcal{B}, \mathcal{C}$ is an isomorphism from $\mathcal{B}$ to $\mathcal{C}$, which maps the ith generator of $\mathcal{B}$ to the ith generator of $\mathcal{C}$, for $i = 1..n$. A structure $\mathcal{A}$ is *homogeneous* if every isomorphism between two n-generated substructures of $\mathcal{A}$ extends to an automorphism of $\mathcal{A}$, for $n \in \mathbb{N}$.

It is straightforward to verify that the structures $(\mathbb{N}, =)$ and $(\mathbb{Q}, \leq)$ are homogeneous. Those, and many other examples are discussed in [Mac11].

The following result is well known from model theory (cf. [Hod97]).

Theorem 3.2. *Suppose that $\mathcal{A}$ is homogeneous, over a finite signature and for every n there is a bound on the size of n-generated substructures of $\mathcal{A}$. Then $\mathcal{A}$ is ω-categorical, and each formula is equivalent to a quantifier-free formula.*

The aim of this section is to give an effective version of Theorem 3.2. Observe that if every formula can be *effectively* converted into an equivalent quantifier-free formula, then the theory of $\mathcal{A}$ is decidable. The following lemma is a crucial, though immediate observation, relating homogeneity to ω-categoricity.

Lemma 3.3. *Let $\mathcal{A}$ be a homogeneous structure. If $\bar{x}$ and $\bar{y}$ are two n-tuples of elements of $\mathcal{A}$, which generate isomorphic n-generated substructures of $\mathcal{A}$, then $\bar{x}, \bar{y}$ are in the same orbit.*

To get a good grip on the complexity bounds, we introduce a few notions.

Extension bounds. An *extension* of an n-generated structure $\mathcal{B}$ is an $(n+1)$-generated structure $\mathcal{C}$ whose substructure generated by the first n generators of $\mathcal{C}$ is equal to $\mathcal{B}$. For a structure $\mathcal{A}$, an *extension bound* is a function $e : \mathbb{N} \to \mathbb{N}$ such that for every $n \in \mathbb{N}$, any n-generated structure $\mathcal{B}$ which embeds into $\mathcal{A}$ has at most $e(n)$ non-isomorphic extensions to a structure $\mathcal{C}$ which embeds into $\mathcal{A}$. For example, the 2-generated structure $\mathcal{B} = (\{a, b\}, \leq)$ with $a \neq b$, $a \leq b$, has five (up to isomorphism) extensions to a 3-generated structure which embeds into $\mathcal{Q}$, corresponding to: $c < a < b$, $c = a < b$, $a < c < b$, $a < b = c$, $a < b < c$. For the structures listed above such bounds are, $e(n) = n + 1$ for the pure set, and $e(n) = 2n + 1$ for the rational numbers. If $\mathcal{A}$ has extension bound $e_{\mathcal{A}}$, then for $n \in \mathbb{N}$, let $e!_{\mathcal{A}}(n)$ denote $c_0 \cdot e_{\mathcal{A}}(0) \cdot e_{\mathcal{A}}(1) \cdots e_{\mathcal{A}}(n-1)$, where c_0 is the number of isomorphism types of 0-generated substructures of $\mathcal{A}$. Observe that $e!_{\mathcal{A}}(n)$ is a bound on the number isomorphism types of n-generated substructures of $\mathcal{A}$. This implies:

Lemma 3.4. *Let $\mathcal{A}$ be a homogeneous structure. If $\mathcal{A}$ has extension bound e, then $\mathcal{A}^n$ has at most $e!_{\mathcal{A}}(n)$ orbits. It follows that $\mathcal{A}$ is ω-categorical if and only if it has an extension bound.*

Efficient algorithm. For many homogeneous structures one can implement a data structure allowing to efficiently iterate through all (isomorphism types of) n-generated structures which embed into $\mathcal{A}$, which admits the following operations in amortized constant time (1) proceed to the next (isomorphism type) of an n-generated structure which embeds into $\mathcal{A}$, and (2) extend to the first $(n+1)$-generated structure which embeds into $\mathcal{A}$. Also, testing whether a quantifier-free formula ψ with n variables holds in the current n-generated structure can be done in time $O(|\psi|n)$. If there is such a data structure as described above, we say that there is a *constant-delay* extension enumeration algorithm for $\mathcal{A}$.

Proposition 3.5. *Let $\mathcal{A}$ be a homogeneous structure. Suppose that $\mathcal{A}$ has extension bound $e_{\mathcal{A}}$ and constant-delay extension enumeration algorithm. Then, for a given sentence ϕ, deciding whether ϕ holds in $\mathcal{A}$ can be done in time $O(e!_{\mathcal{A}}(r) \cdot |\phi|^2 r)$, where r is ϕ's quantifier rank.*

Proof. If ϕ is a formula with free variables $x_1, \ldots, x_n$, then let $[\phi]$ denote the set of isomorphism types of n-generated structures $\mathcal{B}$ with generators $b_1, \ldots, b_n$, such that for some embedding α of $\mathcal{B}$ to $\mathcal{A}$, the valuation which maps the variable x_i to $\alpha(b_i)$ satisfies ϕ, i.e., $v, \mathcal{A}, \models \phi$. It follows from homogeneity that this does not depend on the choice of the embedding α.

We show by induction on the structure of a formula ϕ with n free variables, that given an isomorphism type of an n-generated structure $\mathcal{B}$, it can be decided whether $\mathcal{B} \in [\phi]$ in time $O(e(r-1) \cdots e(1) \cdot e(0) \cdot |\phi|)$. Proving this will prove the proposition. Indeed, sentences have 0 free variables, and due to the assumption that $\mathcal{A}$ has no constants, there is only one 0-generated structure, namely the empty structure. Testing whether this structure belongs to $[\phi]$ is equivalent to answering whether $\mathcal{A} \models \phi$.

In the inductive base, we consider predicates $R(t_1, t_2, \ldots, t_k)$, where $t_1, \ldots, t_k$ are terms using function symbols and variables. If $\mathcal{B}$ is an k-generated structure, then testing whether $\mathcal{B} \models R(t_1, t_2, \ldots, k)$ can be done in time linear in $|\phi|$.

Now consider the inductive step. The case when ϕ is of the form $\neg\psi$ or $\phi_1 \vee \phi_2$ is easy. Suppose that ϕ is a formula of the form $\exists x.\psi$. Given a structure $\mathcal{B}$, to test whether $\mathcal{B} \in [\phi]$, consider all extensions $\mathcal{B}'$ of $\mathcal{B}$ by one generator, and find out whether one of them satisfies $\mathcal{B}' \in [\psi]$. This can be done in the required time, by inductive assumption. □

The internal solver of LOIS uses the procedure from Proposition 3.5 for several theories of homogeneous structures (see [KTc] for more details).

4 Tests

We have tested LOIS with its internal solver, as well as with two state-of-the-art SMT solvers conforming to the SMT-LIB standard, namely CVC4 [BCD+11] and Z3 [DMB08]. We have also tested the solver SPASS, which is based on the superposition calculus [WDF+09]. In the tests, the underlying structure was $(\mathbb{Q}, \leq)$, which has the same first-order theory as $(\mathbb{R}, \leq)$. For the external solvers, we used the LRA logic (Linear Real Arithmetic), which is the weakest logic defined in the SMT-LIB 2 standard which encompasses the theory of $(\mathbb{R}, \leq)$. Six LOIS programs were used as benchmarks: testing basic properties of orders, reachability and three minimisation algorithms. These examples arise naturally from our motivations, discussed in Section 5. The results are presented in Figure 1. The tests indicate that there is space for improvement for state-of-the-art SMT solvers in performing quantifier elimination in formulas which do not involve arithmetic. In particular, the *order* test, which is a simple program testing transitivity of the linear order on $\mathbb{Q}$, is surprisingly difficult for external solvers. See [KTa] for an archive containing the generated queries, command line options, and other details.

	order	*reachable*	*minimize1*	*minimize2*	*minimize3*
internal	0.0: 0	0.0: 0	0.5: 0	33.8: 0	1.2: 0
Z3-4.3.2	7.4: 12	0.6: 1	5.0: 7	158.9: 229	2.7: 1
CVC4-1.4	0.1: 51	0.1: 11	3.8: 478	58.4: 241	9.3: 2
CVC4-1.4*	0.1: 85	3.7: 67	18.3: 57	*hangs*	10.3: 2
SPASS-3.5	110.6:107	3.7: 0	111.6: 0	905.9: 1076	256.1: 1
queries	159	180	8732	5962	28616

Figure 1: Results of tests. Columns correspond to tests, rows to solvers. An entry of the form $t : u$ means that the test took t seconds, and that to u queries the solver replied "unknown". The last row shows the total number of queries. In CVC4-1.4*, finite model finding is enabled.

5 Applications

This section serves as an illustration of the potential applications of LOIS to formal verification. The point in case is that LOIS provides a new bridge between SMT solvers and formal verification. We give some examples of classes of infinite-state systems known from formal verification, which can be naturally modeled using definable sets, and that verification problems can be solved using simple LOIS algorithms.

Definable automata. Fix an underlying logical structure $\mathcal{A}$. A *definable automaton* is defined just as a nondeterministic finite automaton (NFA), but all its components are required to be definable over $\mathcal{A}$, rather than finite – the statespace Q, the alphabet Σ, the transition relation $\delta \subseteq Q \times \Sigma \times Q$, the initial and final states $I, F \subseteq Q$. The automaton from Example 1.1 is a definable automaton over $(\mathbb{N}, +, \leq, 0)$, and also over $(\mathbb{N}, +)$, as $\leq$ and 0 are definable using $+$.

Definable automata can be presented as input for algorithms, by using the expressions which define them, and in LOIS, simply by using definable sets. As in automata theory, a central problem in verification to which many problems reduce is the *reachability problem*: does a given automaton have an accepting run?

Example 5.1. *Register automata* of Kaminsky and Francez [KF94] are (roughly) finite-state automata additionally equipped with finitely many registers which can store data values from an infinite set D, and which process sequences of data values from D. In each step, basing on the current state and equality or inequality tests among the values in the registers and the current input value, the automaton can choose to store the current value in one of its registers (replacing the previous value), change its state, or continue to the next input value. For example, we could consider a register automaton with two registers recognizing the set of those sequences $d_1 d_2 \dots d_n \in D^*$ such that $d_n \in \{d_1, d_2\}$. It is not difficult to prove [KF94] that the reachability problem for register automata is decidable (in fact, in PSPACE).

Register automata are a special case of definable automata, where the underlying structure $\mathcal{A}$ is $(D, =)$, or equivalently, $(\mathbb{N}, =)$. Indeed, if a register automaton has m states and n registers, then the corresponding definable automaton has statespace $Q = \{q_1, \ldots, q_m\} \times \mathcal{A}^n$ (we treat $q_1, \ldots, q_m$ as symbols; cf. Remark 2.2), and input alphabet $\Sigma = D$. The transition relation $\delta \subseteq Q \times \Sigma \times Q$ is a definable set over $(D, =)$ as it is defined only using equalities and inequalities.

Many other models of infinite-state systems can be naturally viewed as special cases of definable automata, over a suitably chosen structure $\mathcal{A}$. This includes rational relational automata of Cerans [ACJT96] (underlying structure $(\mathbb{Q}, \leq)$), vector addition systems [ACJT96] (VASs, related to Petri nets – underlying structure $(\mathbb{Z}, \leq, 0)$), timed automata [AD94] (underlying structure $(\mathbb{R}, \leq, 0, +1)$), database driven systems [Via09] (the underlying structure is a generalization of the Rado graph, see [BST13]), Fraïssé automata [BKL14] (the underlying structure is the limit of a Fraïssé class of finite structures), and many others.

For any definable automaton one can run the LOIS reachability algorithm described in Example 1.1. It is clear that the algorithm is correct, i.e., it will produce the right output, whenever it terminates. Therefore, two things remain: to provide an SMT solver for the theory of $\mathcal{A}$, and to prove termination. There are two generic principles of proving termination of the reachability algorithm for infinite-state systems from the literature: the first one involves well-quasi orders (this includes VASs, and is discussed broadly in [FS01]), and the second one involves ω-categoricity (this includes register automata, and is discussed discussed in [FH12]). Below, we briefly mention the latter.

The following result is a simple consequence of ω-categoricity.

Proposition 5.2. *The procedure* `reach` *from Example 1.1 terminates whenever the sets $E \subseteq V \times V$ and $I \subseteq V$ are definable over an ω-categorical structure $\mathcal{A}$.*

Together with Theorem 2.3, this yields the following.

Theorem 5.3. *Reachability is decidable for all definable automata over a fixed ω-categorical underlying structure with decidable theory.*

This result implies decidability of the reachability problem for register automata from Example 5.1, rational relational automata, and many others, generalizing slightly the results from [BKL11, BT12, BKL14]. Note that thanks to LOIS and definable sets, a single algorithm can be used, and no specific data structures are needed to prove decidability for a wide class of infinite-state systems. As a consequence, the termination proof can focus on the mathematical content, and not on the specifics of the implementation.

On a side note, we remark that an analysis of the proof of Proposition 5.2, together with the results described in Section 3, yield optimal (PSpace) complexity bounds for the models mentioned above. Finally, the ω-categoricity principle also yields sound and complete procedures for other problems concerning infinite-state systems studied in the verification literature, e.g., reachability for definable pushdown automata [MRT14] and definable tree automata (defined in the natural way), or the minimization of definable automata, using the procedure presented in Example 1.1.

6 Related work

The idea of a programming language which allows working with infinite sets, thus providing a useful tool in verification and in automata theory, was proposed by Bojańczyk et al. [BBKL12] – who proposed a functional language called Nλ – and Bojańczyk and the second author [BT12] – who proposed an imperative language. Differences between LOIS and these languages include the semantics (cf. [KTb]), but most importantly, the fact that they are restricted to (some) homogeneous structures over finite signatures, and do not employ SMT solvers, whereas LOIS allows handling any structure for which an SMT solver is provided.

Declarative programming paradigms offer some form of manipulation of infinite sets. In logic programming and constraint programming, predicates and constraints are typically infinite. In functional

programming, the programmer manipulates functions as first-class objects. Furthermore, lazy evaluation allows performing operations on infinite streams. However, our approach is fundamentally different, as it represents sets internally by formulas, allowing to effectively scan through infinite set. In particular, membership and equality of sets can be effectively tested. On the other hand, we can only handle *definable* sets. In fact, both approaches – functional programming and manipulating infinite sets – are orthogonal, and can be combined, as proposed in Nλ [BBKL12], and implemented in [KS] using definable sets. We remark that [JKS12] is yet another, orthogonal extension of functional programming by the ability of testing equality between certain infinite sets, namely regular coinductive datatypes, and uses equation solvers for this purpose.

Superficially, LOIS is similar to Kaplan [KKS12] – an extension of the Scala programming language. Its main purpose is to integrate constraint programming into imperative programming. It allows effective manipulation of constraints, and relies on a verification tool Leon, which in turn invokes the SMT solver Z3. Constraints are implemented as boolean valued functions (in Scala, functions are first-class objects) whose arguments are integers or algebraic data types built on top of integers. As such, they can be seen as certain logical formulas which can be defined as programs in a fragment of the Scala language. However, this fragment is incomparable with first order logic, as it allows recursion but not quantification. More importantly, the main objective of LOIS – to allow iterating over infinite sets – is not addressed in Kaplan (one can perform list comprehension in order to iterate through the explicit set of solutions of a constraint, which terminates only if this set is finite). It would be interesting to see whether iteration over infinite sets defined by constraints can be incorporated into Kaplan.

SMT solvers have been applied in various branches of formal verification [BSST09, DMB11, AMP06, DMB14]. In particular, in model checking they are used in predicate abstraction, interpolation-based model checking, backward reachability analysis and temporal induction. LOIS offers yet another application of SMT solvers in model checking: to analysis of infinite-state systems.

References

[ACJT96] Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, and Yih-Kuen Tsay. General decidability theorems for infinite-state systems, 1996.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[AMP06] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. In Antti Valmari, editor, *Model Checking Software, 13th International SPIN Workshop, Vienna, Austria, March 30 - April 1, 2006, Proceedings*, volume 3925 of *Lecture Notes in Computer Science*, pages 146–162. Springer, 2006.

[BBKL12] Mikołaj Bojańczyk, Laurent Braud, Bartek Klin, and Sławomir Lasota. Towards nominal computation. In Field and Hicks [FH12], pages 401–412.

[BCD+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[BKL11] Mikołaj Bojańczyk, Bartek Klin, and Sławomir Lasota. Automata with group actions. In *LICS*, pages 355–364. IEEE Computer Society, 2011.

[BKL14] Mikołaj Bojańczyk, Bartek Klin, and Sławomir Lasota. Automata theory in nominal sets. *Log. Meth. Comp. Sci.*, 10, 2014.

[BSST09] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, pages 825–885, 2009.

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. Available at `www.SMT-LIB.org`.

[BST13] Mikołaj Bojańczyk, Luc Segoufin, and Szymon Toruńczyk. Verification of database-driven systems via amalgamation. In Richard Hull and Wenfei Fan, editors, *PODS*, pages 63–74. ACM, 2013.

[BT12] Mikołaj Bojańczyk and Szymon Toruńczyk. Imperative programming in sets with atoms. In Deepak D'Souza, Telikepalli Kavitha, and Jaikumar Radhakrishnan, editors, *FSTTCS*, volume 18 of *LIPIcs*, pages 4–15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[DHPV09] Alin Deutsch, Richard Hull, Fabio Patrizi, and Victor Vianu. Automatic verification of data-centric business processes. In *Intl. Conf. on Database Theory (ICDT)*, 2009.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340. Springer-Verlag, 2008.

[DMB11] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, 54(9):69–77, September 2011.

[DMB14] Leonardo De Moura and Nikolaj Bjørner. Applications of smt solvers to program verification. Available online: http://fm.csl.sri.com/SSFT14/smt-application-chapter.pdf, 2014.

[FH12] John Field and Michael Hicks, editors. *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*. ACM, 2012.

[FS01] Alain Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001.

[Hod97] Wilfrid Hodges. *A Shorter Model Theory*. Cambridge University Press, New York, NY, USA, 1997.

[JKS12] Jean-Baptiste Jeannin, Dexter Kozen, and Alexandra Silva. CoCaml: Programming with coinductive types. Technical Report http://hdl.handle.net/1813/30798, Computing and Information Science, Cornell University, December 2012. Fundamenta Informaticae, to appear.

[KF94] Michael Kaminski and Nissim Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, 1994.

[KKS12] Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. Constraints as control. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 151–164, New York, NY, USA, 2012. ACM.

[KM69] Richard M. Karp and Raymond E. Miller. Parallel program schemata. *J. Comput. Syst. Sci.*, 3(2):147–195, May 1969.

[KS] Bartek Klin and Michał Szynwelski. *SMT Solving for Functional Programming over Infinite Structures*. Submitted.

[KTa] Eryk Kopczyński and Szymon Toruńczyk. *LOIS* – online repository containing the LOIS prototype, its source code and tests. See http://www.mimuw.edu.pl/~erykk/lois/.

[KTb] Eryk Kopczyński and Szymon Toruńczyk. *LOIS: syntax and semantics*. See http://www.mimuw.edu.pl/~erykk/lois/.

[KTc] Eryk Kopczyński and Szymon Toruńczyk. *LOIS: technical documentation*. See http://www.mimuw.edu.pl/~erykk/lois/.

[Mac11] Dugald Macpherson. A survey of homogeneous structures. *Discrete Mathematics*, 311(15):1599–1634, 2011.

[MRT14] Andrzej S. Murawski, Steven J. Ramsay, and Nikos Tzevelekos. Reachability in pushdown register automata. In Erzsébet Csuhaj-Varjú, Martin Dietzfelbinger, and Zoltán Ésik, editors, *Mathematical Foundations of Computer Science 2014 - 39th International Symposium, MFCS 2014, Budapest, Hungary, August 25-29, 2014. Proceedings, Part I*, volume 8634 of *Lecture Notes in Computer Science*, pages 464–473. Springer, 2014.

[Via09] Victor Vianu. Automatic verification of database-driven systems: a new frontier. In *Intl. Conf. on Database Theory (ICDT)*, pages 1–13, 2009.

[WDF+09] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. Spass version 3.5. In *Proceedings of the 22Nd International Conference on Automated Deduction*, CADE-22, pages 140–145, Berlin, Heidelberg, 2009. Springer-Verlag.