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 demonstrate how 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 [KTa] 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; as a result, the set X 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 [KTb]. 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 $\exists x.(x > 3) \wedge (2 \cdot x + 1 = 10)$ in $(\mathbb{N}, \cdot, +)$.

We believe that LOIS offers an extremely 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. 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 #. We use pseudocode similar to LOIS syntax to the left below, and comment it to the right.

```
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))
```

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 the letter # 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.

```
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 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.
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 $\{(m, n) \mid m, n \in \mathbb{N}, m \geq n\}$.

We 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 # equals 3. 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 theunique initial state.

```
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
}
```

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 [KTb]), with no need of auxiliary data structures.

Syntax and semantics. The syntax and semantics of LOIS are very intuitive, yet novel. However, this is not the topic of this paper, but of our companion paper [KTa]; the C++ library is described in [KTb].

SMT solvers. What is relevant for this paper is 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 (cf. Section 5). For instance, in the example above, the underlying theory is the theory of the structure $(\mathbb{N}, +, \leq, 0)$ which is used for defining the particular automaton.

Contributions. The aim of both this paper and its companion [KTa] is to introduce various aspects of LOIS. Whereas [KTa] defines a formal syntax and semantics and proves results about them, this paper concentrates on applications of LOIS to formal verification and the use of SMT solvers. We also discuss the implementation of several SMT solvers which are crucial for our applications, i.e., for ω-categorical theories admitting quantifier elimination. We argue that LOIS provides a new application of SMT solvers to formal verification.

Outline. A central notion underlying LOIS are *definable sets*, introduced in Section 2. We argue that LOIS has many potential applications to algorithm design in formal verification, as it allows to avoid unnecessary implementation details by directly manipulating infinite mathematical objects. In Section 3 we give many examples to illustrate this, including register automata [KF94], rational register automata [ACJT96], Petri nets [KM69], timed automata [AD94], pushdown register automata [MRT14], database driven systems [Via09], context-free languages over infinite alphabets [CK98].To implement these examples, LOIS resorts to SMT solvers for theories which have not been of central focus in the SMT community, in particular, ω-categorical theories admitting quantifier elimination. We implement simple solvers for several such theories (see Section 4), and show that they outperform state-of-the-art SMT solvers by orders of magnitude (cf. Section 5). We discuss the related work in Section 6 and conclude in Section 7.

2 Definable sets and LOIS

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

Definable sets. The sets which can be manipulated by LOIS programs have a purely logical description. We refer to the literature (e.g. [Hod97]) for structures, sorts, terms, and formulas. In this paper, formulas are assumed to be first-order.

Fix an infinite *underlying* logical structure $\mathcal{A}$, which may involve relation or function symbols. For simplicity in the definition below, we assume that $\mathcal{A}$ has only one sort named $\mathbb{A}$; generalizing to multisorted structures is straightforward. 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 an 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. 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 the finite unions of open (possibly half-bounded) intervals and points. The set of all open intervals in $\mathcal{Q}$ is a $\emptyset$-definable set, as 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 3. 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. Finally, we will sometimes 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. We briefly describe the LOIS language. See the companion paper [KTa] for the formal semantics, and [KTb] for the exhaustive list of constructions available in our implementation. 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 $\mathbb{A}, \mathbb{B}, \ldots$ of type `set`, one for each sort of $\mathcal{A}$,
- 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 $\mathcal{A}$.

We discuss some parts of the semantics of LOIS in Section 5.1, with an emphasis on how SMT solvers are employed.

Remark 4. In the companion paper [KTa] 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++). The implementation of the tool [KTb] follows very closely the semantics of the language, so in this paper, we will take the liberty to use ambiguous phrases such as "LOIS computes a set".

A structure $\mathcal{A}$ 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 $(\mathbb{N}, +, \cdot)$ has not, by Gödel's theorem. In this paper the underlying structure $\mathcal{A}$ is always assumed to have decidable theory.

Definable sets are the central data structure underlying LOIS programs, used to represent elements of type `set`. They have very good closure properties – they are effectively closed under boolean combinations, cartesian products, projections, quotients, intersections and unions of definable families, 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 [KTa].

Theorem 5. *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*.*

3 Applications

This section serves as an illustration of the potential applications of LOIS to formal verification. We give many examples of classes of infinite-state systems known from formal verification, which can be naturally modeled using definable sets. We also show LOIS algorithms which can be used for solving various problems for those classes of systems, and present several termination proofs. As an important case, we distinguish ω-categorical structures.

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 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 6. 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 $Q = \{q_1, \dots, q_m\} \times \mathcal{A}^n$ ($q_1, \dots, q_m$ are treated as symbols), and input alphabet $\Sigma = D$. The transition relation $\delta \subseteq Q \times \Sigma \times Q$, as it is defined only using equalities and inequalities, is a definable set over $(D, =)$.

Example 7. Rational relational automata of Cerans (see [ACJT96]) are similar to register automata, but process sequences of rational numbers rather than elements of D, and can base their decisions on comparisons with respect to the linear order. These automata are a special case of definable automata, where the underlying structure is $(\mathbb{Q}, \leq)$, and the definitions involve quantifier-free formulas only. Rational register automata also have a decidable reachability problem, with the same complexity (PSpace) as register automata.

Example 8. A *Minsky machine* is a model equipped with several counters storing natural numbers, which can be incremented, decremented, and tested for zero.

The result of Minsky is that reachability for his machines is undecidable. Minsky machines are also a special case of definable automata, over the structure $(\mathbb{N}, +1)$.

Example 9. Consider the coverability problem for *Vector Addition Systems* (VASs, related to *Petri nets* – see e.g. [ACJT96]), defined below. Fix a dimension $k \geq 0$. A *VAS* of dimension k is described by a finite set of vectors $V \subseteq \mathbb{Z}^k$. We say that a vector t is *coverable* from s if, starting from s, one can reach a vector componentwise larger than t, by repeatedly adding vectors from V so that the intermediate result always stays in the positive fragment, $\mathbb{N}^k \subseteq \mathbb{Z}^k$. It is known that the coverability problem is decidable for VASs [KM69,FS01].

A VAS gives rise to a definable automaton over the underlying structure $\mathcal{Z} = (\mathbb{Z}, <, 0)$, as follows. For a fixed finite set $V \subseteq \mathbb{Z}^k$ there is a formula ϕ_V with $2k$ free variables such that $\phi_V(u, w)$ holds iff $w \leq u + v$ for some $v \in V$; the formula ϕ_V refers to the relations $<, =$ only, and not to addition or subtraction. Therefore, a VAS induces an automaton with states $Q = \mathbb{N}^k$, transition relation

$$\delta = \{(u, w) \mid u, w \in W, \phi_V(u, w)\},$$

initial states $I = \{u \in W \mid u \leq s\}$ and accepting states $F = \{v \in W \mid v \geq t\}$; this automaton is definable over $\mathcal{Z}$. In fact, it is a definable *well-structured transition system*, using the terminology of [FS01]. It is easy to see that this system has an accepting run if and only if t is coverable from s.

Example 10. Timed automata [AD94] can be seen as a special case of definable automata, with underlying structure $(\mathbb{R}, \leq, 0, +1)$. The input alphabet is of the form $\{a_1, \ldots, a_r\} \times \mathbb{R}$; a word over this alphabet is called a timed word if the timestamps are increasing. If a timed automaton has m states and n clocks, then the corresponding definable automaton has states $\{q_1, \ldots, q_m\} \times \mathbb{R}^n$, and has a transition relation defined using quantifier-free formulas of restricted form: boolean combinations of atomic formulas $c_i \leq t + k$ (*timed constraints*) or $c_i = 0$ (*reset*), where c_i is the ith clock and t is the timestamp read on input. Reachability of timed automata is decidable using the so-called *region construction*.

Fix an underlying structure $\mathcal{A}$. Every definable automaton can be constructed in LOIS, similarly to Example 1. Therefore, for any definable automaton one can run the standard reachability algorithm described in Example 1. It is clear that the algorithm is correct, i.e., it will produce the right output, whenever it terminates. The aim is therefore to identify classes of definable automata for which the algorithm does terminate. This is a purely mathematical question: the programming part is done, since the reachability algorithm is executable in LOIS, assuming an SMT solver for the underlying structure is provided.

3.1 Proofs of termination

We now present several termination arguments for the models listed above. As a side effect, we prove Theorem 14 and Theorem 20, i.e., decidability of the reachability problem for automata and pushdown automata which are definable over *ω-categorical* structures with decidable theory, generalizing known results.

Termination for ω-categorical structures. We now present a generic termination argument for the reachability algorithm from Example 1, which works for all definable automata over a wide range of underlying structures $\mathcal{A}$, including register automata from Example 6, rational relational automata from Example 7, and many others, generalizing slightly the results from [BKL11,BT12,BKL14]. This argument uses the notion of *ω-categoricity* from model theory.

For a structure $\mathcal{A}$, its *automorphism* is a bijection of $\mathcal{A}$ to itself, which preserves the sorts, 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 11. The structure $(\mathbb{N}, =)$ is ω-categorical. Its automorphisms are all bijections of $\mathbb{N}$ to itself. Representatives of the orbits of $\mathbb{N}^3$ are $(1,1,1), (1,1,2)$, $(1,2,1), (2,1,1), (1,2,3)$. More generally, the orbits of $\mathbb{N}^k$ correspond to partitions of the set $\{1, \ldots, k\}$. The structure $(\mathbb{Q}, \leq)$ is also ω-categorical. Its automorphisms are the increasing bijections of $\mathbb{Q}$. Representatives of the orbits of $\mathbb{Q}^2$ are $(1,1), (1,2), (2,1)$. In general, the orbits of $\mathbb{Q}^k$ correspond to transitive, reflexive and total relations on $\{1, \ldots, k\}$. More generally, any *homogeneous* structure over a finite relational signature (see Section 4) is ω-categorical. The structures $(\mathbb{Z}, \leq), (\mathbb{N}, +1), (\mathbb{R}, +1)$ are not ω-categorical: each pair $(0, n)$, for $n \in \mathbb{N}$, represents a different orbit.

Lemma 12. *Let $\mathcal{A}$ be an ω-categorical structure. Then any S-definable set V has only finitely many S-definable subsets.*

We consider the reachability algorithm described in Example 1, which is implemented in LOIS [KTb].

Corollary 13. *The reachability algorithm terminates if the sets $E \subseteq V \times V$ and $I \subseteq V$ are definable over $\mathcal{A}$.*

Proof. Let $S \subseteq \mathcal{A}$ be a finite set such that E, V, I are S-definable. By Theorem 5, in the nth iteration of the loop in the `reach` function, the variable `R` evaluates to an S-definable set $R_n \subseteq V$, giving rise to an increasing sequence:

$$R_0 \subseteq R_1 \subseteq R_2 \subseteq \ldots. \tag{2}$$

By Lemma 12, the sequence (2) stabilizes after finitely many steps. □

Corollary 13 and Theorem 5 imply the following.

Theorem 14. *Reachability is decidable for all definable automata over a fixed ω-categorical underlying structure with decidable theory.*

This result generalizes the decidability of the reachability problem for the models described in Examples 6 and 7, and by a slightly more detailed analysis which we omit here, yields optimal complexity bounds in each of those cases. Theorem 14 also generalizes slightly a result from [BKL11,BKL14].

Example 15. Database driven systems [DSVZ06,Via09,DHPV09] extend register automata by the capability of testing whether a tuple of data values stored in the registers belongs to a relation of a database $\mathbb{D}$ over a fixed schema Σ. The reachability problem is: given a description of a system $\mathcal{S}$, does there exist a finite database $\mathbb{D}$ over the schema Σ, such that $\mathcal{S}$ has an accepting run driven by $\mathbb{D}$? It is observed in [BST13] that this is equivalent to: does $\mathcal{S}$ have an accepting run, driven by the *infinite* database $\mathbb{D}_\infty$? The structure $\mathbb{D}_\infty$ a generalization of the Erdös-Rado graph [Rad64]; we define it when Σ consists of a binary relation E, interpreted as an edge relation, the general case is similar. The structure $\mathbb{D}_\infty$ has a countable set of vertices, and for each pair of vertices v, w independently and at random with probability $1/2$, an edge (v, w) is created in $\mathbb{D}_\infty$. With probability 1, two independent constructions of $\mathbb{D}_\infty$ result in isomorphic structures. Hence $\mathbb{D}_\infty$ is called *the* random directed graph. It is homogeneous (cf. Section 4), in particular, ω-categorical, and has decidable theory.

A database-driven system $\mathcal{S}$ defines a definable automaton over $\mathbb{D}_\infty$. Theorem 14 yields decidability of reachability of this automaton, which is equivalent to reachability of the original system $\mathcal{S}$. Hence, reachability for database-driven systems is decidable (this gives the optimal PSPACE bound for a fixed schema Σ). Note that this proof is substantially simpler than the original proof [DSV07] (existence of infinite runs can be reduced to the case of finite ones as in [ST11]).

Using the above method, one can also prove decidability in the case when the databases $\mathbb{D}$ are XML trees conforming to a fixed XML schema; it is then useful to consider *infinite homogeneous trees* [BST13] instead of $\mathbb{D}_\infty$.

Other termination arguments. Termination of the reachability algorithm can be also proved when the underlying structure is not ω-categorical, for some classes of definable automata.

Example 16. To test coverability of a VAS, convert it into a definable automaton as described in Example 9. The reachability algorithm from Example 1 terminates and correctly decides reachability of this automaton, thus deciding coverability of the original VAS by using an SMT solver for $(\mathbb{N}, \leq)$. The termination argument is the well known [FS01] well quasi-order argument: the sequence (2) stabilizes by Dickson's lemma, as it consists of downward-closed subsets of $\mathbb{N}^k$ (with respect to the coordinatewise ordering of $\mathbb{N}^k$). More generally, the reachability algorithm terminates for any definable well-structured transition system.

Example 17. The reachability algorithm terminates for timed automata (see Example 10), treated as definable automata. To prove termination, one argues that each set R_n in the sequence (2) decomposes into finitely many regions. To run this algorithm in LOIS, an SMT solver for $(\mathbb{R}, \leq, +1, 0)$ is needed.

Note that by using LOIS, no specific data structures need to be devised: the algorithm is the standard one from Example 1, and is trivial to implement in LOIS (see [KTb]). Thanks to this, the termination proofs can be more abstract and precise, since they do not need to discuss the implementation details. This luxury of abstraction can be afforded thanks to SMT solvers, which are heavily used when LOIS instructions are executed (see Section 5).

3.2 Other verification problems

So far we only investigated one problem: the reachability problem for definable automata. Many other classical algorithms involving finite-state systems can be executed in LOIS on infinite-state definable systems, often yielding interesting theoretical decidability results. We illustrate this phenomenon with two further examples: automata minimization and reachability of pushdown automata.

Example 18. Deterministic definable automata are defined just as finite deterministic automata, i.e., they are nondeterministic definable automata in which the transition relation is the graph of a function $\delta : Q \times \Sigma \to Q$. The automaton from Example 1 is a deterministic definable automaton. The standard Moore minimization algorithm presented in Example 1 can be run for any definable deterministic automaton; it is immediate to implement it in LOIS [KTb]. It follows from Corollary 13 that this algorithm terminates whenever the underlying structure is ω-categorical. The algorithm also terminates for many other automata, such as the one given in Example 1. It is shown in [BL12] that minimization terminates for deterministic timed automata, represented appropriately.

Example 19. Definable automata can be extended to *definable pushdown automata*, just as normal pushdown automata extend NFAs. Here, we extend the automata by ε-transitions, add to the syntax a definable stack alphabet Γ, and require the transition relation to be a definable subset of $Q \times (\Sigma \cup \{\varepsilon\}) \times Ops \times Q$, where $Ops = \{push, pop\} \times \Gamma$ is the set of stack operations. For the underlying structures $(\mathbb{N}, =)$, this model generalizes *pushdown register automata* [MRT14].

We implement in LOIS a standard fixpoint algorithm for pushdown automata which computes the set of pairs (p, q) such that there is a run from state p to state q which starts and ends with an empty stack; this algorithm relies on the following branching analogue of the reachability procedure from Example 1.

```
function deduce(rules) {
  set R = ∅;
  bool added = true;

  while (added) {
    added = false;
    for (premises,conclusion)
    in rules do
      if ((premises⊆ R)
        and (conclusion∉R)) {
          R += conclusion;
          added = true; }
  }
  return R;
}
```

The function `deduce` takes as an argument a set of rules of the form (X, x), where X is a subset of some set of facts F and $x \in F$, and computes the least $R \subseteq F$ such that $X \subseteq R$ implies $x \in R$ for every rule (X, x). For infinite definable arguments over ω-categorical structures, the function `deduce` terminates for exactly the same reason as in the proof of Corollary 13, as a consequence of Lemma 12.

Using the above algorithm (which is executable in LOIS, despite infiniteness of the arguments!), we prove the following theorem (see Appendix C.1 for details).

Theorem 20. *Reachability is decidable for all definable pushdown automata over a fixed ω-categorical underlying structure with decidable theory.*

An analogous result can be proved for *definable tree automata* with infinitely many states and infinite alphabet, or for infinite, *definable grammars*. Similar results were described in [BKL11,BBKL12,BT12,BKL14,CL15] (see Section 6).

All the above examples serve as a proof of concept, to demonstrate that infinite-state systems can be seamlessly manipulated using a programming language manipulating definable sets, which in turn relies on SMT solvers (we discuss this in Section 2). As a consequence, proofs of termination (and also complexity bounds, see Remark 25) become more abstract and concise. Two paradigms are particularly useful for termination proofs: the ω-categoricity argument and the well quasi-ordering argument (see [FS01] for a broad overview), but other combinatorial arguments can be also employed.

By using LOIS, one can come up with many other algorithms for problems involving classes of infinite-state systems. LOIS acts as an intermediary which converts imperative programs into queries to SMT formulas, as discussed in Section 5. As far as we know, this provides a new use of SMT solvers in verification.

4 Theories of homogeneous structures

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 3. In this section, we discuss the algorithm briefly. In Section 5 we discuss how and when the internal solver and external solvers are invoked in LOIS, and how they perform in tests.

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 i-th generator of $\mathcal{B}$ to the i-th 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. Also, the countable directed random graph $\mathbb{D}_\infty$ defined in Section 3 is homogeneous. Those, and many other examples are discussed in [Mac11].

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

Theorem 21. *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 21. 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. Due to lack of space, the proofs in this section are relegated to Appendix C.

Lemma 22. *Let $\mathcal{A}$ be a homogeneous structure. Suppose that $\bar{x}$ 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 of the action of $\mathrm{Aut}(\mathcal{A})$ on $\mathcal{A}^n$.*

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 the following property holds 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, respectively: for the pure set $e(n) = n + 1$, for the rational numbers $e(n) = 2n + 1$, for the random graph $e(n) = n + 2^n$. 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 23. *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 24. *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.*

Remark 25. Under assumptions as in Proposition 24, S-definable subsets of V in Lemma 12 can be iterated in polynomial space with respect to the size of the description of V. This implies that the reachability algorithm for definable automata runs in polynomial space, giving optimal bounds for register automata, rational relational automata, and others. Also, this implies (an optimal) exponential running time for the reachability algorithm for pushdown automata.

5 Solvers and tests

The only prerequisite for LOIS to work with an underlying structure $\mathcal{A}$ is that a solver for its (first-order) theory is provided. LOIS invokes the solver for the theory of $\mathcal{A}$ in order to determine whether a given instruction should be executed (in `for`, `if`, `while` statements). In this section, we give some details about the used internal and external solvers.

5.1 Queries generated by LOIS

We briefly describe how LOIS generates formulas. This is part of the formal semantics described in detail in our companion paper [KTa].

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 by using `if`.

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, LOIS can use its internal solver, which tests satisfiability using the algorithm described in Proposition 24, and is implemented for some theories of homogeneous structures, including the examples from Section 4, and several others (cf. [KTb]). We compare the performance of the internal and external solvers in Section 5.2.

Origin of the formulas. It is perhaps worth expounding on the origin 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 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 in the expected way. For example, $X \subseteq Y$ is implemented by the code below. It returns a formula with the expected quantifier prefix $\forall^*\exists^* \ldots$. In general, tests for $\in, =, \subseteq$ between sets of nesting n produces a formula with $\leq 2n$ 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.

5.2 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 superposition calculus [WDF+09]. In the tests, the underlying structure was $\mathcal{Q}$ (rational numbers with order); it has the same 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 from the tutorial and three minimisation algorithms (different automaton than in the introduction). More details on the performed tests are provided in Appendix B.3. The results are presented in Figure 1.

	order	*reachable*	*minimize1*	*minimize2*	*minimize3*
internal	7: 0	3: 0	549: 0	33884: 0	1215: 0
Z3	7368: 12	569: 1	5022: 7	158928: 229	2734: 1
CVC4	76: 51	121: 11	3807:478	58395: 241	9257: 2
CVC4*	113: 85	3729: 67	18303: 57	*hangs*	10320: 2
SPASS	110582:107	3666: 0	111567: 0	905887:1076	256187: 1
queries	159	180	8732	5962	28616

Fig. 1. Results of tests. Columns correspond to tests, rows to solvers. An entry of the form $t : u$ means that the test took t milliseconds, and that to u queries the solver replied "unknown". The last row shows the total number of queries.

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.

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, comes from the papers [BBKL12] – which proposes a functional language called Nλ – and [BT12] – where an imperative language is proposed. Both languages are capable of handling infinite, but *orbit-finite sets with atoms*. These papers were meant as a proof of concept, and did not propose any efficient implementation. In contrast, we present a working prototype. Similarly to this paper, they show applications of such languages in formal verification, which form a subset of the applications of LOIS, since they are limited to homogeneous underlying structures over finite relational signatures (which are ω-categorical by Theorem 21).

Furthermore, they propose to represent infinite sets internally as finite unions of orbits. This has several drawbacks. Firstly, the underlying structure $\mathcal{A}$ needs be homogeneous and over a finite signature, whereas LOIS can work with any structure with decidable theory. Secondly, the orbit decompositions proposed in those papers are exponentially less concise than expressions describing definable sets, rendering them impractical for most applications, in particular, optimal complexity results cannot be obtained this way. Lastly, SMT solvers cannot be easily employed. An implementation of Nλ in Haskell using definable sets and SMT solvers is the topic of our sister project [KS].

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.

For a discussion of the novel syntax and semantics of LOIS see [KTa].

7 Conclusion

We introduce an imperative programming language which allows seamless manipulation of infinite sets, by employing SMT solvers. In this paper, we focused on the applications to verification of infinite-state systems, and aspects involving the use of the solvers. LOIS is implemented as a C++ library, and several algorithms for infinite-state systems are presented in this paper and implemented in LOIS [KTb]. LOIS uses an internal solver for theories of ω-categorical homogeneous structures. There are many improvements possible in this area, for example, improving the internal solver and extending it to other theories.

We made a case that LOIS provides a new connection between formal verification and SMT solvers, different from the various connections established so far [DMB14]. We would like to encourage both communities to benefit from this connection. In particular, the SMT community might consider using queries produced by LOIS as benchmarks, whereas the formal verification community might benefit from using LOIS as a convenient language for algorithm design, owing to its ease of representation of abstract mathematical objects.

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

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 John Field and Michael Hicks, editors, *POPL*, pages 401–412. ACM, 2012.

[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.

[BL12] Mikołaj Bojańczyk and Sławomir Lasota. A machine-independent characterization of timed languages. In Artur Czumaj, Kurt Mehlhorn, Andrew Pitts, and Roger Wattenhofer, editors, *Automata, Languages, and Programming*, volume 7392 of *Lecture Notes in Computer Science*, pages 92–103. Springer Berlin Heidelberg, 2012.

[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.

[CK98] Y.C. Edward Cheng and Michael Kaminski. Context-free languages over infinite alphabets. *Acta Informatica*, 35(3):245–267, 1998.

[CL15] L. Clemente and S. Lasota. Reachability analysis of first-order definable pushdown systems. In *Proc. CSL'15*, pages 244–259, 2015.

[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.

[DSV07] Alin Deutsch, Liying Sui, and Victor Vianu. Specification and verification of data-driven web applications. *J. Comput. Syst. Sci.*, 73(3):442–474, 2007.

[DSVZ06] Alin Deutsch, Liying Sui, Victor Vianu, and Dayou Zhou. A system for specification and verification of interactive, data-driven web applications. In *Intl. Conf. on Management of Data (SIGMOD)*, 2006.

[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.

[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: syntax and semantics*. Submitted for a conference. See `http://www.mimuw.edu.pl/~erykk/lois/`.

[KTb] 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.

[OEI] The on-line encyclopedia of integer sequences. http://oeis.org.

[Rad64] R. Rado. Universal graphs and universal functions. *Acta Arithmetica*, 9:331–340, 1964.

[ST11] Luc Segoufin and Szymon Toruńczyk. Automata based verification over linearly ordered data domains. In Thomas Schwentick and Christoph Dürr, editors, *28th International Symposium on Theoretical Aspects of Computer Science, STACS 2011, March 10-12, 2011, Dortmund, Germany*, volume 9 of *LIPIcs*, pages 81–92. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[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.

A Details of tests

In this section, we provide the details of the tests we have been running on LOIS. The sources are included in the subdirectory `tests` in the LOIS package.

A.1 Automatic tests

The program `tests/autotest.cpp` performs some automatic testing of LOIS. This includes some interesting applications of LOIS, allowing one to see that LOIS runs correctly, and how fast does it run. The following tests are conducted:

- `testRandomBipartite`
  Let R be a random symmetric and anti-reflexive relation, and let A_1 and A_2 be a random 2-partition. Let $S(x, y)$ iff x and y are in different parts of the partition. We construct a new relation $E = R \cap S$. In graph theoretic terms, R is Rado's random graph, S is a complete bipartite graph, and E is a random bipartite graph. We take one vertex $x \in \mathbb{A}$ and run BFS on the graph, and ask about the number of iterations after which we have reached every vertex. The program correctly answers that every vertex is reached after 3 iterations.
  This test evaluates in roughly 3 milliseconds on the machine used for tests.
- `testTree`
  A function is given elements $x_1, \ldots, x_k$ of the homogeneous tree, and asks questions about relationships between them. Once the answers uniquely determine the substructure generated by $x_1, \ldots, x_k$, the substructure is presented in a readable form. For four elements without any relations, 416 possible structures are generated (262 if we know that all the four elements are not equal — see sequences A005264 and A005172 in [OEI]).
  This test evaluates in roughly four seconds on our machine (for four elements). This time is relatively long because of two reasons:
  - The extension bound of the homogeneous tree is relatively large ($e(n) = 8n - 4$, which gives the evaluation time of roughly $8^k k!$ according to Proposition 24). In fact Proposition 24 is not optimal, all trees are generated in time roughly linear in the number of all trees, which is 416 for four elements. Still, it grows quite fast.
  - The program is very ineffective: currently, each question tries to generate all the possible structures from the beginning, even if we know that some possible structures have been already ruled out.

  Therefore, the running time is actually at least quadratic in the number of possible trees. This should be optimized in the future versions of LOIS.
- `testOrder`
  This test the basic properties of the order relation, and evaluates very quickly.
- `testAssigment`
  This checks whether an assignment exception is correctly thrown when we try to assign a value (`rbool`, in this case) which uses variables which are not in the internal context of the variable we are assigning to.

- `testQueue`
  This checks whether the `setof`'s and the queue semantics of the `for` loop works correctly. Numbers from 0 to 10 are inserted to `lsetof<int>`.
- `testRemoval`
  This checks whether the `-=` operator works in the natural, pseudo-parallel way, as advertised in paper.

A.2 Minimisation of an automaton

The program `tests/mintest.cpp` tries to perform the minimisation algorithm on an orbit finite automaton. This automaton over the alphabet $\mathbb{A}$ (our infinite domain) reads three symbols, and accepts iff either two of them are equal (if there are less than three or more than three symbols, the word is rejected). The minimisation algorithm works in a way similar to the usual one for finite automata. The equivalence relation $\eta \subseteq Q \times Q$ is computed – two states will be in η if they can be merged into a single state. Initially, η is set to $F \times F \cup (Q - F) \times (Q - F)$, and then, in each iteration states each $x, y \in Q$ are separated iff $\neg\eta(\delta(x, a), \delta(y, a))$ for some symbol a in the alphabet. For this particular automaton minimisation takes four iterations. The algorithm is implemented using two representations (η is represented either as a relation or as the set of equivalence classes), and currently takes 0.15 s in the relation representation, and 32 s in the equivalence class representation.

A.3 Solver tests

The program `tests/soltest.cpp` tests various solvers on several LOIS functions. The table in Section 5.2 is based on its results. The following tests are included:

- `testOrder` This test the basic properties of the order relation, and evaluates very quickly with the internal solver, although external solvers have problems with it.
- `testReachable` Reachability from the introduction.
- `testReal` This test the basic properties of the `Real` sort (LRA logic).
- `testInt` This test the basic properties of the `Real` sort (LRA logic).
- `testMinimizeXY` These tests minimize automata. There are two automata: A (the same as in `tests/mintest.cpp`) and B (the automaton using the integers from the introduction), and three different implementations of the minimisation algorithm (two from `tests/mintest.cpp`, and the one shown in the Introduction is implementation number 3). Internal solver and SPASS work on the automaton A, but none of the solvers work on B.
- `testPacking` What are the maximal sets of subsets of (0,5) such that no two points are in distance less than 1? Z3 correctly calculates using the LRA logic that such maximal sets can have from 3 to 5 elements.
- `testCirclePacking` This tests the NRA logic by asking about packings of disks in a larger disk. None of the tested solvers can answer even the simplest questions here.

The external solvers where called using the following commands:

Z3: `z3-*/bin/z3 -smt2 -in -t:500`
CVC4: `cvc4 -lang smt -incremental -tlimit-per=500`
CVC4*: `cvc4 -lang smt -incremental -finite-model-find -tlimit-per=500`
SPASS: `SPASS -TimeLimit=1`

B Proofs

In this appendix, we present some proofs of the results from Section 3 and Section 4.

B.1 Proofs for Section 3

Proof of Theorem 20.

Proof. We implement a standard fixpoint algorithm for pushdown automata which computes the set U of pairs (p, q) such that there is a run from state p to state q which starts and ends with an empty stack.

```
function isEmptyPushdown
           (Σ,Q,Γ,δ,I,F) {
  set rules = ∅;

  for q in Q do
    rules += (∅,(q,q));

  for (p,q,r) in Q³ do
    rules+= ({(p,q),(q,r)},(p,r));

  for (u,a,(push,γ),p) in δ do
  for (q,b,(pop,γ),v) in δ do
    rules+= ({(p,q)},(u,v));

  return (deduce(rules)∩(I×F))!=∅
}

function deduce(rules) {
  set R = ∅;
  bool added = true;

  while (added) {
    added = false;
    for (premises,conclusion)
    in rules do
      if (premises⊆ R)
        and (conclusion∉R)
          R += conclusion;
          added = true;
  }
  return R;
}
```

For this, we apply two kinds of deduction rules (we write $p \to q$ if $(p, q) \in U$):

$$p \to q,\ q \to r \implies p \to r,$$
$$p \to q \implies u \to v,$$

whenever $(u, a, \mathrm{push}_\gamma, p), (q, b, \mathrm{pop}_\gamma, v) \in \delta$.

The function `deduce` takes as an argument a set of rules of the form (X, x), where X is a subset of some set of facts F and $x \in F$, and computes the smallest set $R \subseteq F$ satisfying $X \subseteq R \implies x \in R$ for every rule (X, x).

The function `deduce` terminates for exactly the same reason as in the proof of Corollary 13: the subsequent values of R are increasing, and are subsets of the set of facts F, which is S-definable, where $S \subseteq \mathcal{A}$ is a finite set such that `rules` is S-definable.

Correctness of the procedure `isEmptyPushdown` is clear. This proves Theorem 20.

B.2 Proofs for Section 4

Proof of Theorem 21.

Proof. By Lemma 22, the orbit of $\bar{x} \in \mathcal{A}^n$ is described by the quantifier-free formula with n free variables describing the isomorphism type of the (finite) n-generated structure generated by $\bar{x}$. In particular, there are finitely many such orbits. A formula ϕ with n-free variables defines a union of orbits of $\mathcal{A}^n$, so it is equivalent to a disjunction of quantifier-free formulas.

Proof of Lemma 22.

Proof. By homogeneity, the isomorphism extends to an automorphism of $\mathcal{A}$, so $\bar{x}, \bar{y}$ are in the same orbit of the action of $\mathrm{A}ut(\mathcal{A})$ on $\mathcal{A}^n$.

Proof of Lemma 23.

Proof. From Lemma 22 it follows that the orbit of an n-tuple $\bar{x}$ is determined by the isomorphism type of the structure it generates. This proves the first part of Lemma 23, and the 'if' implication of the second part. For the 'only if' implication, note that $e(n)$ is bounded by the number of orbits of $\mathcal{A}^{n+1}$.

Proof of Proposition 24.

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 $\mathcal{A} \models \phi$.

We proceed to the inductive proof. 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 $|\phi|$.

In the inductive step, consider a formula ϕ 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 case when ϕ is of the form $\neg\psi$ or $\phi_1 \vee \phi_2$ is easy.